Composition and Interoperability for External Domain-Specific Language Engineering

Thomas Degueule

ANNÉE 2016

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Thomas Degueule

préparée à l'unité de recherche IRISA (UMR 6074)
Institut de Recherche en Informatique et Systèmes Aléatoires
ISTIC – UFR en Informatique et Électronique

Composition and Interoperability for External Domain-Specific Language Engineering

Thèse soutenue à Rennes le 12 décembre 2016

devant le jury composé de :

Mark van den Brand
Professeur, Eindhoven University of Technology / Rapporteur

Richard Paige
Professeur, University of York / Rapporteur

Sandrine Blazy
Professeur, Université de Rennes 1 / Examinatrice

Ralf Lämmel
Professeur, University of Koblenz-Landau / Examinateur

Bernhard Rumpe
Professeur, RWTH Aachen University / Examinateur

Olivier Barais
Professeur, Université de Rennes 1 / Directeur de thèse

Arnaud Blouin
Maître de conférences, INSA Rennes / Co-directeur de thèse

Benoit Combemale
Maître de conférences, Université de Rennes 1 / Examinateur

Acknowledgments

First and foremost, I would like to thank all the members of my jury for accepting to review this manuscript and for attending my defense. I am extremely grateful and honored by their presence. In particular, I would like to thank Mark van den Brand and Richard Paige for their thorough review of my manuscript and their precious feedback. I would also like to thank Sandrine Blazy, Ralf Lämmel, and Bernhard Rumpe who accepted to act as examiners. Believe it or not, I could hardly have dreamed of a better jury. Presenting my work in front of you was a rewarding and pleasant experience, and I really enjoyed the discussion that followed. Thank you for your kind and perceptive comments!

It goes without saying that this thesis is not only the result of my work, but also that of my awesome trio of supervisors: Olivier Barais, Arnaud Blouin, and Benoit Combemale. Olivier, thank you for your unconditional support and your constant cheerfulness. Throughout these three years, it felt good receiving your daily dose of positivity. Arnaud, thank you for all the time you spend on the many papers we wrote, dealing with my terrible English writing skills. Thank you also for your ~~debatable~~ sense of humor. May *torgen* rule them all. Benoit, thank you for being such a great scientific mentor. You taught me most of the few things I know about our research field – and research in general. You always took the time to introduce me to the people who matter, not as a student but as a colleague. I truly realize how rare it is for a PhD student to receive so much time and help from his supervisors. The three of you always supported me in every occasion, and I am honored by the unreserved confidence you placed in me throughout these three years. I know that the end of my thesis does not imply the end of our collaboration, and I look forward to working with each of you again in the future!

Besides my direct supervisors, I am grateful to the DiverSE team as a whole. It has been extremely rewarding to work in a team that gathers so many awesome researchers, covers so many subjects, and, most importantly, constitutes such a pleasant and lively research environment. I know that some

people thank the heads of their team and laboratory because they are politically obliged to do so. It is instead a pleasure for me to thank Benoit Baudry and Jean-Marc Jézéquel, not because I have to, but because they inspired me a lot, played an important role in the work presented here and, most crucially, in shaping my future research. Regarding my other friends from DiverSE, I must confess that I am way too scared to write down an explicit list of names here, as I would hate myself if I forget anyone. How would I have sorted this list anyway? I thank each and every one of them, as they all played a different but important role in these three years. I cannot stress this enough:

DiverSE is an amazing team in every way.
You guys are amazing.
Thank you.

Of course, I do not forget that the first persons who led me to the realm of research during my master's thesis are Jean-Marie Mottu and Gerson Sunyé. Is there any better way to start research than by killing mutants? I do not think so. Had I not met such kind and supportive researchers, I would probably never have pursued on this path. Besides, I am entirely beholden to you for my arrival in DiverSE. Thank you!

One can easily consider Melange – or was it K3SLE? ;) – as one of the main contributions of this thesis. As such, all the people involved in Melange deserve a place here. In particular, I would like to thank the awesome scientific engineers who actively worked on its implementation. Thank you Fabien for all the great stuff you did, especially for maturing Melange from the ever-buggy research prototype I was developing to an actually usable software project. Thank you Didier for all the time you spent integrating our stuff with the other software platforms of the team. I really enjoyed working with both of you, and you truly played an important role in the results presented in this thesis. ~~I apologize in advance for all the time you will spend maintaining and developing Melange in the future.~~

I do not really feel comfortable with the idea of thanking my family and friends here – they will never read these lines anyway. Rather than on a piece of paper, I express my gratitude to them through every moment we share.

Contents

Chapter 1

Introduction

In this chapter, I first introduce the context of this thesis (Section 1.1) and detail the research questions I address (Section 1.2). Then, I give an overview of our approach, list the various scientific contributions (Section 1.3), and the different applications on which we validate them (Section 1.4). Finally, I mention the industrial and international collaborations that motivated our contributions (Section 1.5) and outline the general organization and the different chapters of this manuscript (Section 1.6).

1.1 Context

The advent of complex software-intensive systems, such as Systems of Systems, Cyber-Physical Systems or the Internet of Things, raises numerous new software engineering challenges. The development of these systems involves many stakeholders, each with diverse areas of expertise, who contribute their knowledge on specific aspects of the system of interest. Typically, stakeholders are accustomed to express their knowledge in the form of *models* using specialized concepts and notations commonly used within their discipline (e.g., mechanical engineering, energy management, systems engineering). An intuitive approach to foster the involvement of stakeholders in the development of such systems is to provide them with appropriate means to express their models of the system, and to use these models actively throughout the development process. The ultimate vision is to break down the barrier between software engineering and other engineering disciplines by enabling domain specialists to actively participate in the development of software systems in which they have an interest.

Model-Driven Engineering (MDE) [231] contributes to this vision in various ways. First, MDE proposes to reduce the accidental complexity associated with the development of complex systems by bridging the gap between the problem-level abstractions used by stakeholders and the implementation-level concepts used to realize the target software system [108]. Second, leveraging the time-honored practice of separation of concerns, MDE advocates the use

of multiple Domain-Specific Languages (DSLs), each with a dedicated set of concepts, notations, and tools suitable for modeling a particular aspect of the system. One key idea of MDE is to move from descriptive models to prescriptive models that are used to construct the target system. Such models are used to perform early verification and validation activities, and automate the generation of essential software artifacts (e.g., components, test cases, documentation). The benefits of MDE range from improvements in the productivity of developers to enhanced software quality [137, 12].

Recent studies show that the use of appropriate DSLs is one of the key factor for the successful adoption of MDE in industrial settings [138, 273]. DSLs aim at bridging the gap between the problem space (in which stakeholders work) and the solution space (the concrete software artifacts defining the target system) [107]. They are usually small and intuitive software languages whose concepts and expressiveness are focused on a particular domain [191]. As "software languages are software too" [97], the development of DSLs and their supporting environment (editors, generators, simulators, etc.) inherits the complexity of software development in general. Because the domains they focus are continuously evolving, DSLs tend to evolve at a faster rate than general-purpose languages. It follows that concerns such as reusability, maintainability or evolution must be taken into account when engineering them. Fortunately, good practices from the software engineering community can be leveraged to achieve these engineering goals. The need for proper tools and methods supporting the development of DSLs led to the emergence of the Software Language Engineering (SLE) research field which Kleppe defines as "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages" [159].

On the one hand, MDE relies on SLE techniques for efficient and maintainable engineering of DSLs. On the other hand, MDE proposes a solid theory of language definition for the definition of DSLs. MDE and SLE thus cross-fertilize each others, and this thesis is positioned at the intersection of these two research fields.

1.2 Problem Statement

DSLs exist in different shapes, ranging from fluent application programming interfaces [105] to external DSLs. Notably, external DSLs offer the best flexibility and added value for the end users but are the most costly to engineer as they cannot rely on the infrastructure of a host language [106]. The development of an external DSL encompasses the definition of its abstract syntax, concrete syntax, semantics, and a dedicated environment that assists users of the language in the creation, analysis, and management of the conforming models. Following the principles of MDE, the development of a new DSL usually starts with the definition of a metamodel, the cornerstone artifact that specifies its

abstract syntax. Concrete syntaxes and semantics are then specified as mappings from the abstract syntax to the appropriate co-domains [123]. Finally, the environment accompanying a DSL consists of a set of integrated services (e.g., transformations, checkers) that analyze, manipulate, and transform the conforming models.

The definition of each of these artifacts is prone to errors and requires substantial engineering efforts [153, 264]. Moreover, using today's technologies, the same process is typically repeated from scratch for every new DSL or new version of a DSL. To foster their adoption in the industry, the benefits in terms of productivity when using DSL technologies must offset the initial investment required in developing such DSLs. Therefore, tools and methods must be provided to assist language designers in the development of new DSLs and the evolution of legacy ones to mitigate development costs. Techniques for increasing reuse from one DSL to another and supporting the customization of legacy ones to meet new requirements are thus particularly welcomed.

In addition, the proliferation of independently developed and constantly evolving DSLs in numerous areas raises new challenges concerning their integration and the interoperability of their environments [60, 45]. The theoretical foundations of MDE imply that models are tightly coupled with the DSL to which they conform, and modeling environments are tightly coupled with the DSL they support. This has severe consequences on the evolution of DSLs and the interoperability between modeling environments. First, when a DSL evolves, both its conforming models and its environment must be updated accordingly. It is hardly possible to define generic services that manipulate models conforming to different versions of the same DSL. Second, DSL users cannot benefit from services defined for different, yet similar, languages. As a concrete example, it is not possible to open and manipulate a hierarchical statechart model in a modeling environment defined for flat statecharts. This lack of flexibility for the DSL users hinders the sharing of models between different environments, and thereby the collaboration between stakeholders.

From these two observations, we summarize the current limitations imposed on both DSL designers and users as two interrelated challenges:

Challenge #1 The proliferation of independently developed and constantly evolving DSLs raises the problem of interoperability between similar languages and environments. Language users must be given the flexibility to open and manipulate their models using different variants and versions of various environments and services to foster collaboration in the development of complex systems.

Challenge #2 Since DSLs and their environments suffer from high development costs, tools and methods must be provided to assist language designers and mitigate development costs. A promising approach is to

build on the time-honored practices of modular development and reuse in software engineering and adapt them to the specificities of SLE.

1.3 Contributions

To tackle the aforementioned challenges, we structure our approach around three interconnected contributions depicted as ❶, ❷, and ❸ in Figure 1.1.

As a basis for the two other contributions, we propose the notion of *language interface* (❶) as a mean to enhance abstraction and genericity in DSL engineering. Language interfaces allow to abstract the intrinsic complexity carried in the implementation of languages, by exposing meaningful information (i) concerning an aspect of a language (e.g., syntactical constructs) (ii) for a specific purpose (e.g., composition, reuse, coordination) (iii) in an appropriate formalism (e.g., a metamodel). In this regard, language interfaces can be thought of as a reasoning layer atop language implementations. Using language interfaces, one can vary or evolve the implementation of a DSL while retaining the compatibility with the services and environments defined on its interface. Language interfaces enable the definition of generic services that do not depend on a particular version or variant of a DSL and that can be reused for any DSL matching the interface, regardless of its concrete implementation. Our two other contributions, aimed at both language designers and users, rely on this common fundamental notion of language interfaces.

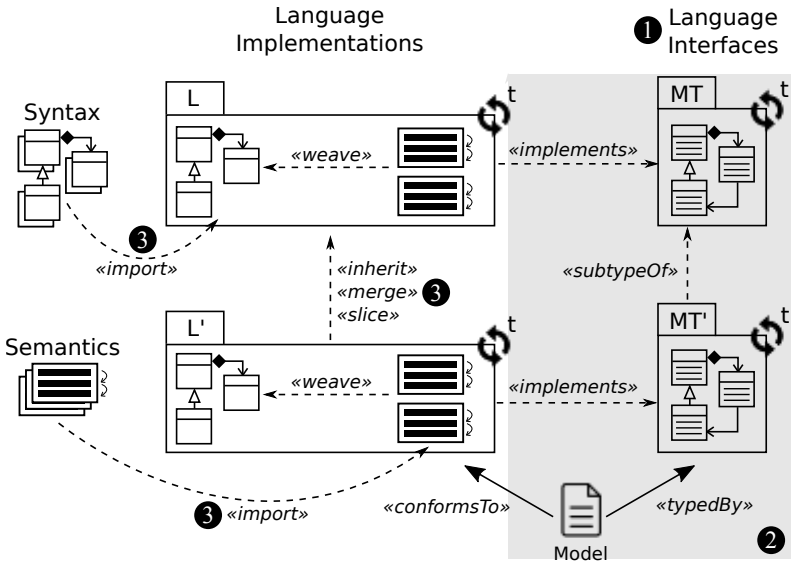


Figure 1.1: Overview of the contributions

To address **Challenge #1**, we propose a disciplined approach that leverages family polymorphism [92] and structural typing [39] to provide an advanced type system for manipulating models, in a polymorphic way, through different

language interfaces (❷). In our approach, these interfaces are captured in model types and supported by a typing relation between models and model types [239] and a subtyping relation between model types [119]. Model types are structural interfaces over a language. They are used to define a set of constraints over admissible models. The subtyping relations define the safe substitutions from one DSL to another, thereby providing more flexibility to DSL users in the manipulation of models conforming to these DSLs. We call this fundamental mechanism *model polymorphism*, i.e., the ability to manipulate a model through different language interfaces. Model polymorphism opens up the possibility to safely manipulate models using different modeling environments and services (e.g., transformations, checkers, compilers).

To address **Challenge #2**, we propose a meta-language for modular and reusable development of DSLs. Many DSLs, despite being targeted at specialized domains, share common paradigms and constructs [218]. When engineering new DSLs, it is thus likely that previous efforts spent on the development of other languages could be leveraged, especially when their domains overlap. For this purpose, we propose an algebra of operators that enables language designers to reuse legacy DSLs, compose them, extend them, and customize them to meet new requirements (❸). This approach relies on the aforementioned dedicated type system to provide a reasoning layer for ensuring the structural correctness of composed DSLs and their safe manipulation.

We implement all our contributions in a new language workbench named Melange.[1] Melange supports the modular and reusable definition of the syntax and semantics of DSLs, and flexible manipulation of the conforming models. Melange is seamlessly integrated with the *de facto* standard Eclipse Modeling Framework (EMF) and provides model polymorphism to any EMF-based tool of the Eclipse modeling ecosystem. We use Melange to evaluate our contributions on various case studies.

1.4 Applications

To validate our contributions and their implementation in the Melange language workbench, we evaluate the capability of Melange to solve challenging language engineering scenarios commonly faced by DSL designers and users.

Our first application shows how the model polymorphism mechanism provided by Melange provides safe and seamless interoperability between structurally similar languages, and compatibility between subsequent versions of the same language (**Challenge #1**). Specifically, we study four versions of the UML metamodel and show that Melange provides the expected flexibility, allowing to open and manipulate the same models using different versions of the UML metamodel. In addition, we show how various model transformations can be reused from one language to other ones, provided that they share some

[1] http://melange-lang.org

commonalities materialized in a common language interface. Specifically, we detail how several transformations, expressed in Java and ATL [150], can be reused for a family of statechart languages consisting of eight variants exposing syntactic and semantic variation points.

In our second application, we use Melange to design a new language for modeling Internet of Things (IoT) systems. Instead of starting its definition from scratch, we use Melange to build the IoT language as an assembly of three publicly available language implementations: (i) the Interface Description Language from the OMG [203] (ii) the Activity Diagram from UML [207] and (iii) the Lua programming language [140]. Results show that using Melange allows to reuse and customize existing language implementations in the creation of new DSLs, thereby taming their high development costs (**Challenge #2**).

1.5 Context of this Thesis

This thesis was conducted through many fruitful interactions with academic and industrial partners. Many ideas leading to the contributions presented here and the Melange language workbench emerged from discussions raised within the collaborative projects ITEA2 MERgE,[2] ANR INS GEMOC,[3] and LEOC Clarity.[4]

Melange was extensively used in the context of the ANR GEMOC project to modularly define executable modeling languages, and support their extension. The Melange language workbench is integrated in the GEMOC studio, an Eclipse package supporting the methodology developed within the GEMOC initiative [31].

In collaboration with Thales Group, we have also experienced Melange in the context of the Clarity project for the definition of extensions and viewpoints on a large-scale systems engineering language named Capella[5].

1.6 Outline

Part I — Background and State of the Art

Chapter 2 introduces the foundations and current practices of MDE and SLE, with particular emphasis on the engineering of DSLs and the role of language workbenches.

Chapter 3 reviews the state of the art of modularity and composition of DSLs, and current support for flexibility and interoperability in the MDE ecosystem.

[2] http://www.merge-project.eu/
[3] http://gemoc.org/ins/
[4] http://www.clarity-se.org/
[5] https://polarsys.org/capella/

Part II — Contributions

Chapter 4 summarizes the purposes of interfaces in software engineering and programming, introduces the notion of language interface, and highlights the benefits of language interfaces for the engineering of software languages.

Chapter 5 presents our contribution on safe model polymorphism, increasing interoperability and flexibility in MDE ecosystems.

Chapter 6 presents our contribution to the modular and reusable development of DSLs through a dedicated meta-language.

Part III — Implementation

Chapter 7 presents the various features of the Melange language workbench through many illustrative examples.

Part IV — Conclusion and Perspectives

Chapter 8 resumes our contributions and identifies the perspectives that directly stem from them.

Part I

Background and State of the Art

Chapter 2

Background

In this chapter, I introduce the theoretical background and main concepts used in the remainder of this thesis. Specifically, I introduce the notion of software language (Section 2.1), and present the main principles of software language engineering (Section 2.2) and model-driven engineering (Section 2.3). I put a particular emphasis on the specificities of domain-specific languages and the role of language workbenches. Along the presentation of these concepts, I draw the boundaries of our contributions.

2.1 Software Languages

Software languages play a primary role in many areas of computer science and software engineering. The term *software language* encompasses all kinds of languages used in the development of software systems: modeling languages, programming languages, configuration languages, markup languages, formal languages, ontologies, etc. [159]. Orthogonally to the different kinds of software languages, one can make a distinction between *General-Purpose Languages* (GPLs) and *Domain-Specific Languages* (DSLs) [255].[1]

A GPL is a software language that can be used in any application domain. Most programming languages (e.g., Scala, Haskell, Go), together with some modeling languages (e.g., the Unified Modeling Language – UML [207]), fall into this category. A prime characteristic of GPLs is that they lack specialized concepts and notations tailored to a specific domain of application. Instead, their expressiveness makes them applicable in many areas. Therefore, while GPLs can be employed to implement any kind of software system, it can be difficult for developers to manually bridge the gap between the problem-level concepts that pertain to the application domain and their implementation in a GPL. By their very nature, GPLs are meant to be used by software experts, and hinder the involvement of domain specialists in the development process.

[1] Some authors distinguish DSLs and DSMLs (Domain-Specific *Modeling* Languages). In this thesis, we consider DSMLs to be a particular class of DSLs, and use the term DSL consistently to designate both.

By contrast, a DSL is a software language whose expressiveness is focused on (and usually restricted to) a particular domain of application [191, 107]. The abstractions and notations provided by a DSL are specifically tailored to its domain of application. It follows that DSLs cannot be used to realize any kind of software systems, but only to solve a particular class of problems. Their main benefit lies in the fact that they allow solutions to be expressed at the level of abstraction of the problem domain. Thus, specifications, models, or programs[2] expressed in a DSL can be directly understood, analyzed, and even manipulated by domain specialists [255]. Programs expressed in a DSL are typically more concise and intuitive, easier to understand, reason about, and maintain [254]. The scope of applicability of DSLs ranges from narrow and specific areas (e.g., the Makefile syntax is dedicated to build automation) to broader domains (e.g., a DSL for statechart modeling may be used in many application domains such as language processing, user interfaces, and computer networks). As of today, DSLs are developed and actively used in countless domains [84]. DSLs are also the fundamental mean for the realization of language-oriented programming [271, 106] or language-oriented modeling [58] as proposed by model-driven engineering (cf. Section 2.3). To foster their use, DSLs rely on a dedicated infrastructure, i.e., a set of tools and services that support the users in writing, analyzing, and manipulating models written in the language. These tools support the development process, help finding errors in the specifications, and automate tedious tasks. However, the design of domain-specific tools, or the adaptation of existing tools to the specificities of a particular DSL is an expensive and error-prone task [261].

An important decision when designing a DSL concerns the "shape" of the resulting language. Language designers can choose to build either an external or an internal DSL.[3] The construction of an *external DSL* encompasses the creation of a new language with its own dedicated infrastructure: editors, compilers, interpreters, analyzers, etc. In such a case, language designers can freely design the syntax and semantics of their language, but must write a complete specification using dedicated formalisms that offer the suitable expressiveness for defining each implementation concern [259]. Since those formalisms are languages intended to specify languages, they are usually known as *meta-languages* and vary according to the chosen technological space or language workbench (cf. Section 2.2.4).

In the case of *internal DSLs*, the principle is to take advantage of the infrastructure already provided by a *host language* [135]. The high-level

[2]It is important to note that, in this thesis, we deliberately do not make any fundamental distinction between a *model* and a *program*, i.e., a *sentence* or *linguistic utterance* expressed in a given software language. Some authors even propose to use the portmanteau "*mogram*" to emphasize this idea [159].

[3]Although the terms "internal" and "embedded" are sometimes used interchangeably, we use the term internal DSL to avoid the confusions sometimes associated with embedding [187]

domain concepts of the DSLs are encoded using the language constructs offered by the host language. Editors, parsers, or compilers of the host language can be reused as is, thus lowering the development costs compared to external DSLs. However, following this approach also implies that the capabilities of an internal DSL are restricted to those of the host language: it must comply with the programming paradigm, the type system, and the tooling provided by the host language. For all these reasons, an appropriate selection of the host language is of crucial importance [221]. Concretely, an internal DSL may take the form of a fluent Application Programming Interface (API) that has the "look and feel" of a distinct language [105], or rely on implementation techniques that introduce new syntactic constructs or "sugar" over a GPL (e.g., using macros [32], desugaring [89], extensible compilers [199], or meta-objets [244]).

At the crossroads of these approaches, other authors argue that language designers should not be forced into choosing a particular shape once and for all, but that the shape of a given DSL should adapt according to particular users and uses [62]. Such DSLs are named *metamorphic DSLs* [1]. In this thesis and the remainder of this section, we focus on external DSLs and simply use the term DSL to refer to them.

2.2 Software Language Engineering

The development of software languages is an historically complex task (see for example the development of the Fortran language which reportedly took 18 man-year [10]). Although the techniques used to engineer DSLs are similar to the ones used to engineer any software language, the development of DSLs carries its own specificities and implementation techniques [238, 264]. "Software languages are software too" [97] and, consequently, the development of software languages inherits all the complexity of software development in general: concerns such as maintainability, reusability, evolution, user experience are recurring requirements in the daily work of software language engineers. Additionally, the development of DSLs encompasses the definition of Integrated Development Environments (IDEs) and services that provide crucial added value to language users but are costly to engineer.

Because DSLs have a rather narrow scope of application and are by definition doomed to evolve with the domain they abstract, the benefits of using them must offset the original effort required to engineer them. Therefore, there is room for applying software engineering techniques to facilitate the development of DSLs. This motivation supported the emergence of the Software Language Engineering (SLE) discipline, which Kleppe defines as as "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages" [159]. Just as traditional GPLs, DSLs are typically defined through three main constituents: abstract

syntax, concrete syntax, and semantics [123]. In this section, we review how SLE supports the definition of these artifacts.

2.2.1 Abstract Syntax

The *abstract syntax* of a software language is its fundamental constituent: it defines its syntactic constructs, independent of their representation [181]. One can say that the abstract syntax of a language defines the "form" of its programs. To be automatically processed by tools, a program or model written in a given language is usually first represented in the form of a data structure called an Abstract Syntax Tree (ASTs): the abstract syntax of a language prescribes the structure of such ASTs. Thus, in a sense, the abstract syntax defines the set of all legal ASTs, i.e., the way programs look like from the point of view of language-supporting tools such as compilers and analyzers.

Usually, a dedicated meta-language is used to specify the abstract syntax of software languages. The two most widely used formalisms for specifying abstract syntaxes are (context-free) *grammars* and *metamodels*. In the following, we mainly present the use of grammars and defer the presentation of metamodels to Section 2.3.1. In the original ALGOL report [11], the syntax of ALGOL was described in a notation now known as the Backus-Naur Form (BNF) that specifies both the abstract syntax of a language and its textual representation (i.e., its keywords). McCarthy later argued that the two (abstract syntax and concrete syntax) should be separated [181]. As of today, BNF and Extended BNF [275] are still heavily used for specifying the context-free grammars of (mainly programming) languages.

While grammars are used for the formal specification and verification of languages, they also play an important role from a software language engineering perspective. They serve as the primary input for compiler-compilers, i.e., programs that generate (part of) compilers from high-level language specifications. As an illustration, parsers generators (e.g., Yacc [148], ANTLR [217]) are programs that can automatically generate a language-specific parser from a grammar specification. The *pgen* component of ASF+SDF [251], for instance, generates parser tables from grammar specifications expressed in SDF [124]. Similarly, Spoofax [155] and MontiCore [168] also derive a set of classes corresponding to the grammar. After parsing, the resulting ASTs are set of objects instance of those classes. Many language workbenches also automatically derive editor support (including e.g., syntax highlighting and document outline) from a grammar specification. Others, such as Xtext [93], integrate grammars and metamodels through a unified formalism. We refer the interested reader to other works for more information on the relation between grammars and metamodels [18, 159, 2, 172]. Klint et al. coined the term *grammarware* to designate grammars and the software that support or use them [160].

2.2.2 Concrete Syntax

While the abstract syntax defines the concepts of a language, the concrete syntax defines how they are represented and manipulated by users of the language. Concrete syntax may be either textual, in which case a program is represented as a sequence of characters, or graphical, in which case the program is represented in a graphical layout of arbitrary symbols (e.g., as for UML diagrams or visual programming languages such as Scratch [222]).

As mentioned in Section 2.2.1, the (E)BNF formalism specifies both the abstract syntax and the textual concrete syntax of a language. Terminal symbols of a grammar define the concrete keywords which users use to build syntactically correct programs, following the grammar rules.

There may be many concrete syntaxes for the same abstract syntax. An example would be the representation of integer addition in concrete programs using either the infix, prefix, or postfix notations. The same concept of integer addition, for instance, may be represented either as `(3 + 4)`, `(+ 3 4)`, or `(3 4 +)` in a textual form.

Many modeling languages are manipulated through a graphical concrete syntax. The UML specifications, for instance, directly specify the notations that must be employed for each UML diagram [204]. In this case, model editing consists in the manipulation of graphical shapes (e.g., boxes, arrows, actors).

2.2.3 Semantics

The role of semantics is to attach *meaning* to the constructs of a language. The formal definition of a language's semantics enables formal reasoning about properties and runtime behavior of the programs written in it. Similarly to formal grammars, formal semantic specifications are employed to automatically generate language-supporting tools such as interpreters, compilers, or type checkers.

On the one hand, the *static semantics* of a language defines additional constraints on the structure of valid programs that cannot easily be expressed in the meta-language describing its abstract syntax. This includes, depending on the language's semantics, checking that every variable declaration is unique within a scope, checking that every identifier is declared before it is used, etc. Type systems (which assign a type to constructs of a program) are also often included into the static semantics of a language.

On the other hand, the *dynamic semantics* of a language specifies the runtime behavior of its programs. There are three prominent approaches to define such dynamic semantics [194]. It is worth noting that they are not mutually exclusive, as each of them provides particular benefits and eases certain kinds of analyses and implementations:

Axiomatic Semantics In *axiomatic semantics*, the semantics of a language is given in terms of predicates over its syntax (*axioms*) [104]. Concretely,

an axiomatic semantics can be specified using e.g., Hoare triples [133]. One can consider that axiomatic semantics describes the properties of a program rather than its precise meaning, and is thus particularly suitable for program reasoning and verification.

Translational Semantics A *translational semantics* defines an exogenous transformation from the abstract syntax of a language to another language whose semantics is well-defined (this includes classic compilers). When the translational semantics constructs mathematical objects (called *denotations*), the term *denotational semantics* is usually preferred [230].

Operational Semantics The *operational semantics* of a language defines the meaning of programs in terms of their *execution*, specified as a sequence of computation steps. Typically, it takes the form of a transition function over program configurations (i.e., states). Operational semantics further refines into *small-step operational semantics* [219] and *big-step operational semantics* (also known as *natural semantics* [151]). In the small-step style, computations are defined one step at a time whereas in the big-step style programs are directly related to the result of their execution.

In this thesis, we focus on languages whose semantics are defined operationally, in a small-step style. Structural Operational Semantics (SOS) [219] and its extensions (e.g., MSOS [195] and I-MSOS [196]) are the prominent formalism used to describe the small-step operational semantics of languages, in the form of an inductive transition system. SOS inspired many frameworks that are able to automatically generate interpreters from a SOS-style specification. For example, Prolog interpreters can be derived from MSOS and I-MSOS specifications [220], AST-based Java interpreters can be derived from DynSem specifications [258], and Maude interpreters can be derived from K specifications [227].

Another popular formalism used to specify formal semantics of languages is *attribute grammars* [212] (as in e.g., LISA [187]). Attribute grammars can be used to compute values along the nodes of an AST to enrich it with semantic information (e.g., for static semantics checking purposes), or to define an operational semantics by computing evaluation results. Examples of environments supporting the definition of languages based on attribute grammars include the Silver system [256] and the JastAdd system for compiler construction [87].

2.2.4 Language Workbenches

The term *language workbench* originates from the seminal work of Martin Fowler [106]. The main intent of language workbenches is to provide a unified environment to assist both language designers and users in, respectively, engineering new DSLs and using them. The idea of language workbench is

however not new. The CENTAUR system [29], for instance, implemented the same idea in the late eighties. Modern language workbenches typically offer a set of meta-languages that language designers use to express each of the implementation concerns of a DSL [259], along with tools and methods for composing and analyzing their specifications. Language workbenches are one of the main medium for innovation in SLE and many state-of-the-art approaches ultimately materialize as features of a language workbench.

As of today, many language workbenches have been developed for various technological spaces: Rascal [161], GME [175], Monticore [168, 117, 167], Spoofax [155, 259, 269], LISA [187], Neverlang [249], ASF+SDF [251], MPS, to name just a few. The interested reader can refer to a recent study of Erdweg et al. for an in-depth study of the features offered by different popular language workbenches [91]. In Chapter 3, we specifically analyze their support for language composition and interoperability.

2.3 Model-Driven Engineering

Model-driven engineering (MDE) is a development paradigm that aims at reducing the accidental complexity in the development of complex software systems [231, 108]. Typically, the development of complex systems involve many stakeholders with diverse backgrounds who contribute their knowledge on specific aspects of the system under development (e.g., performance, security). Accidental complexity arises from the wide gap between the high-level concepts used by stakeholders and the low-level concepts provided by programming languages to realize the final system. MDE helps overcome the gap between the problem space and solution space and tame the complexity of the development of such systems in two main ways:

- MDE advocates separation of concerns through the use of multiple DSLs that provide the appropriate abstractions, notations, and tools for modeling a particular aspect of the system. This way, stakeholders can focus on their particular task without losing the "big picture";
- MDE helps to automatically bridge the gap between low-level and high-level concepts by automatically generating crucial software artifacts (e.g., components, documentation, test cases) from high-level models of the system.

Overall, one key idea of MDE is to move from descriptive models – that merely serve as documentation – to prescriptive models that can be manipulated as first-class entities and used to construct the target system [21, 22]. Such models are used to perform early verification and validation of the system, e.g., through simulation or model checking [54], thus bringing forward the detection of possible errors and reducing their impact. Because models are expressed at

a higher level of abstraction that concrete software artifacts, MDE also eases evolution and maintenance of software systems [8], and foster the integration of different technological spaces [95]. Recent empirical studies show that the benefits of MDE range from improvements in the productivity of developers to enhanced software quality [136, 137, 12].

Because MDE relies on the definition of appropriate DSLs to model different aspects of a system, it proposes fundamental techniques for engineering such languages. The MDE foundations and techniques can be referred to as the *modelware* technological space, by analogy with the *grammarware* technological space. It is worth noting that the two are however not mutually exclusive and can complement each other [274, 213]. In the next sections, we introduce the fundamental concepts of MDE and detail the techniques it provides for engineering DSLs.

2.3.1 Metamodels

An important foundation of MDE is *metamodeling* [8], i.e., the process of engineering *metamodels*. A metamodel is the cornerstone artifact of the definition of a language in MDE [52]. In contrast to grammars, metamodels solely specify the abstract syntax of languages. They materialize the knowledge on a given domain: the appropriate concepts, their properties, and the relations between them. Metamodels are themselves models and can be defined in many different formalisms, e.g., using an entity-relationship diagram (as in AToM^3 [71]), or an object-oriented meta-language such as the Meta-Object Facility (MOF) formalism [204] (as in Kermeta [147]). In this thesis, we focus on object-oriented metamodels as envisioned in some of the seminal work in MDE [23]. An object-oriented metamodel (hereinafter referred to simply as metamodel), uses meta-classes to represent a language's concepts. These meta-classes may contain *properties* which are either simple named *attributes* (e.g., an integer) or *references* towards other meta-classes. Meta-classes can *inherit* one another to materialize concept specialization, and can be *abstract*, meaning that they cannot be instantiated in concrete models.

Figure 2.1 depicts the metamodel of a simple finite-state machine (FSM) language named `MiniFsm` using a class diagram notation. We use this illustrative language throughout this section to illustrate the concepts of abstract syntax, concrete syntax, and semantics of a DSL in the modelware space. The metamodel consists of three meta-classes which correspond to the three concepts of the language: finite-state machines (`FSM`), states (`State`), and transitions (`Transition`). A finite-state machine is composed of a set of states (one of which being its initial state) and transitions. A state is identified by its name, and a transition is fired when it receives a particular event. A transition links a source state to a target state. Conversely, a state is associated to the list of its incoming and outgoing transitions.

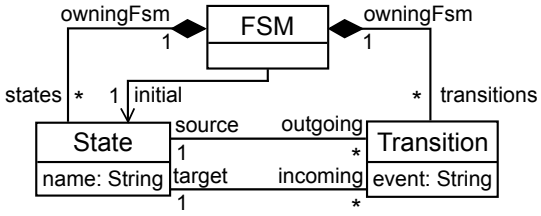


Figure 2.1: Metamodel of the `MiniFsm` language

Metamodels do not always offer the appropriate expressiveness to specify all the structural constraints of a language. In Figure 2.1, it is for example not possible to specify that the initial state must not have any incoming transition. To cope with this limitation, one can complement a metamodel with *static semantics rules*. The static semantics defines well-formedness rules that must hold for all the models conforming to the metamodel, and that cannot be expressed in the formalism used to describe a metamodel. They are typically expressed in the form of pre-conditions, post-conditions, and invariants. Listing 2.1 shows how to specify an invariant on top of the `Transition` meta-class of Figure 2.1 in the Object Constraint Language (OCL) [209], a widely used language for expressing queries and predicates over object-oriented metamodels such as the UML and MOF.

```
context Transition inv: self.target <> self.owningFsm.initial
```

Listing 2.1: An OCL invariant specifying that a `Transition` cannot target the initial state. The `<>` operator checks for object inequality.

The Essential Meta-Object Facility (EMOF) [204], standardized by the Object Management Group (OMG), is a widely used standard for object-oriented metamodeling in the modeling community. EMOF is a subset of the MOF specification that is easily amenable to implementation in concrete tools and integrated with OCL [209]. On the technological side, the Eclipse Modeling Framework (EMF) proposes the Ecore meta-language which is closely aligned with EMOF [240]. Additionally, the EMF ecosystem contributes numerous generative tools around the Ecore language such as editors and model transformation languages (cf. Section 2.3.3). Therefore, EMF is the *de facto* technological standard both in industry and academia: many popular tools such as ATL [150], Xtext [93], Kermeta [147], and Epsilon [162] are built on top of – or interoperable with – EMF and Ecore.

2.3.2 Models

From the description of metamodels given in Section 2.3.1, we give in this section a precise description of models. In the MDE paradigm, a model

conforms to a metamodel if each object in the model is an instance of a concrete meta-class of the metamodel, and if the model satisfies the static semantics rules of the metamodel [24]. The relation that links models to metamodels is called the *conformance relation*. For the sake of conciseness, we can simply write that a model conforms to a language, which means that the model conforms to the metamodel defining the abstract syntax of this language. The characteristics and properties of the conformance relation are the main motivation for one of our contribution. Thus, we refer the reader to Chapter 5 for an in-depth analysis of the properties and limitations of the conformance relation from theoretical and experimental points of view.

Building on the `MiniFsm` metamodel depicted in Figure 2.1, Figure 2.2 depicts a model conforming to the `MiniFsm` metamodel in the form of an object diagram [204]. The object diagram makes explicit the instantiation relations between objects of the model and meta-classes of the metamodel. A model is thus a set of objects (a graph of objects) where each object is instance of exactly one meta-class of the metamodel. Each object instantiates concrete values for the properties defined in its meta-class. By analogy with the terms used in the programming language community, a model is an Abstract Syntax Tree (AST) whose structure is prescribed by the metamodel it conforms to.

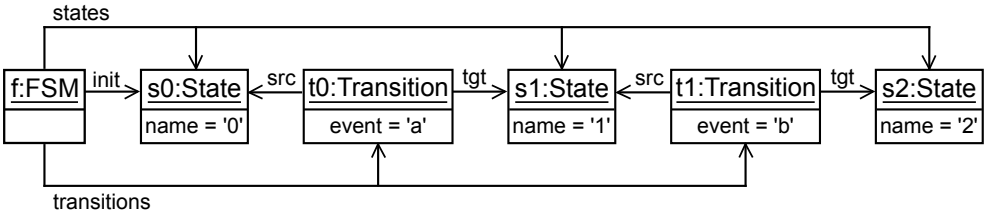
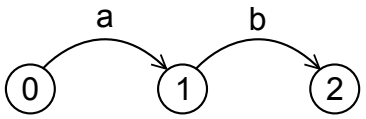


Figure 2.2: A model conforming to the metamodel depicted in Figure 2.1

Naturally, it is not intuitive for language users to directly manipulate models in their abstract form. To support the work of language users, language designers may define one of several concrete syntaxes of their language. These concrete syntax may be graphical, textual, or a mix thereof. Figure 2.3 depicts two alternative concrete representations for the model depicted in Figure 2.2. The choice of a concrete syntax is influenced by the language users' culture and habits. On the technological side, the EMF ecosystem includes frameworks for defining the textual syntax (e.g., Xtext [93]) or graphical syntax (e.g., Sirius [85]) of languages.

2.3.3 Model Transformations

Model transformations play a crucial role in MDE and as such have been identified as the "heart and soul" of MDE [235]. Model transformations are programs that are specifically dedicated to the manipulation of models. They are essential to MDE as they allow to automate recurring modeling



(a) Graphical representation

```
1 statemachine Example {
2   initial state 0
3   state 1
4   state 2
5   transition a from 0 to 1
6   transition b from 1 to 2
7 }
```

(b) Textual representation

Figure 2.3: Two concrete representations of the model depicted in Figure 2.2

activities such as model refactoring [277], slicing [28], code generation and many more [68, 186].

A model transformation takes as input an arbitrary number of *input models* and produces as output an arbitrary number of *output models* (if any). For the sake of clarity however, we limit our presentation to model transformations that accept a single input model and produce a single output model in the remainder of this section. Similarly to functions, procedures, and methods in programming languages, the input and output models of a model transformation are typed by their respective metamodels (named *input and output metamodels* of the transformation). To be processed by a given transformation, a model must *conform* to its input metamodel, and possibly an additional set of transformation pre-conditions. The transformation then ensures that the produced model (if any) conforms to its output metamodel and post-conditions.

Model transformations may be implemented using any programming language and, by extension, any programming paradigm. However, most of the languages used to implement model transformations fall into one of the following paradigms: imperative (e.g., Java, Kermeta [147]), declarative (e.g., QVT-R [211], VIATRA [64]), triple graph grammars [166, 232], or hybrid (e.g., ETL [164], ATL [150]). Model transformation languages may even be derived directly from an existing language, so that they incorporate linguistic abstractions that are tailored to a specific domain of application [134].

Model transformations are usually classified into three categories [186]:

Exogenous transformations are transformations whose input metamodel and output metamodel differ. Examples of transformations falling into this category are transformations used to implement the translational semantics of a language, or integration transformations that transform models from one formalism to the other;

Endogenous transformations are transformations whose input metamodel and output metamodel are the same. Examples of transformations falling into this category are model refactoring and refinement transformations;

In-place transformations are a particular kind of endogenous transformations whose input and output models are the same. An in-place transformation directly modifies its input model "in place" and returns it as output.

As an illustrative example, Listing 2.2 depicts a simple in-place model transformation for the `MiniFsm` language written in ATL, containing one simple transformation rule `AddSuffix`. It takes as input a model conforming to the metamodel depicted in Figure 2.1 and produces as output an equivalent model where all state names are prefixed with "state".

```
1 module atlexample;
2 create OUT : MiniFsm from IN : MiniFsm;
3
4 rule AddSuffix {
5   from sourceState: MiniFsm!State
6   to   targetState: MiniFsm!State (
7     name <- 'state' + sourceState.name
8   )
9 }
```

Listing 2.2: A simple in-place ATL transformation module

In particular, in this thesis, we use model transformations to define the operational semantics of languages. Such transformations are in-place transformations that take a model in a given configuration and perform computation steps over it. Following the good practice of separation of concerns in language definition [259], the semantics of the `MiniFsm` language is defined separately from its syntax, using a dedicated meta-language. As an example, Listing 2.3 shows how to define the different computation steps defining the operational semantics of `MiniFsm` using K3, an imperative meta-language mainly used for specifying the operational semantics of languages (cf. Chapter 7 for a complete presentation of K3). Following the Kermeta approach, computation steps are defined as methods that are woven in corresponding concepts of the abstract syntax using aspects [147], similar to the open class mechanism [57]. One of them is chosen as the entry point for execution.

```
1  // Insert the new runtime concept "current state" in FSM
2  // and weave a new method execute() used as entry point
3  @Aspect(className = FSM)
4  class ExecutableFsm {
5    State current
6    def void execute() {
7      /* Not shown for the sake of conciseness */
8    }
9  }
10 // Weave a new method execute() on State
11 @Aspect(className = State)
12 class ExecutableState {
13   def void execute() {
14     /* Not shown for the sake of conciseness */
15   }
16 }
17 // Weave a method fire() on Transition that
18 // updates the current state of the containing FSM
19 @Aspect(className = Transition)
20 class ExecutableTransition {
21   def void fire() {
22     _self.owningFsm.current = _self.target
23   }
24 }
```

Listing 2.3: Weaving the computation steps defining the operational semantics of `MiniFsm` using K3 aspects

Model Transformation Footprinting When applying a model transformation on a given model, it is likely that it does not access all the elements contained in the model, but only a subset. The subset of the elements accessed or modified by a model transformation is called its *model footprint* [143]. Just like other analysis techniques, model footprinting may be realized either statically or dynamically, as depicted in Figure 2.4.

The *dynamic model footprint* of the application of a model transformation on a given model is the set of model elements that are effectively manipulated during the execution of the transformation. Inferring the dynamic model footprint requires to execute the model transformation, and collect the accessed elements along its execution (e.g., by instrumenting the transformation or analyzing its execution trace). Footprinting can also be used at the metamodel level. The *static metamodel footprint* of a model transformation is the set of metamodel elements required to write the transformation. This information can be extracted by analyzing the definition of a model transformation. From the static metamodel footprint of a model transformation, one can derive its *static model footprint* on a given model, which is the set of all model elements

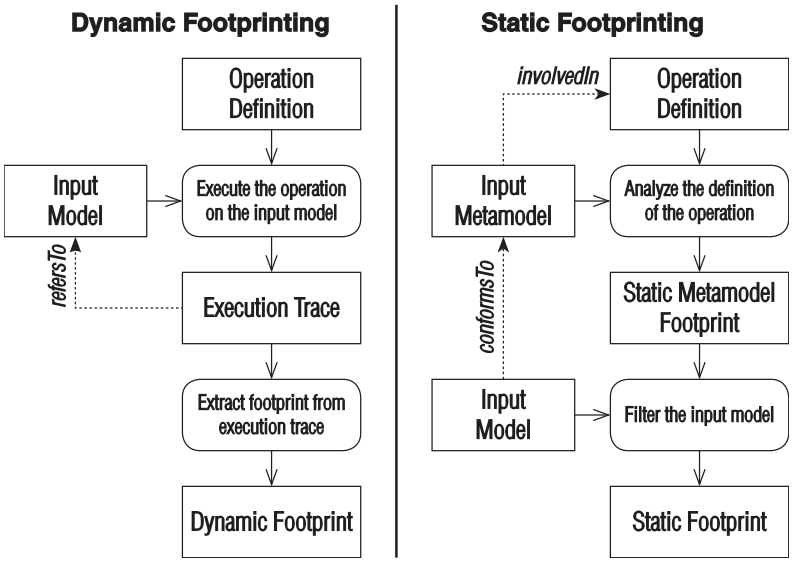


Figure 2.4: Static and dynamic model transformation footprinting, as proposed by Jeanneret et al. [143]

that are instances of the meta-classes found in the static static metamodel footprint.

Footprints of model transformations have multiple uses. They have for instance been used to distribute model transformations across clusters [16], modularize model transformations [66], and implement model slicers [27]. In this thesis, we use model transformation footprinting to acquire knowledge on the syntactic elements manipulated by a given computation step of the semantics of a language, and to infer the elements accessed and modified by tools and services defined around a given language (cf. Chapter 5).

2.4 Summary

In this chapter, we introduced the main concepts used in the remainder of this thesis. Software languages have a long and rich history in computer science, and many techniques have been proposed for supporting their definition. We identified the key foundations on which we build our contributions: we focus on domain-specific languages, their abstract syntax is defined by a metamodel written in an object-oriented meta-language such as MOF, their operational semantics is defined using in-place model transformations expressed for instance in the K3 meta-language, and their concrete syntax takes an arbitrary form.

Chapter 3

State of the Art

In this chapter, I detail a comprehensive state of the art of modularity and composition capabilities in software language engineering, and flexibility and interoperability in model-driven engineering. To this end, I review both theoretical and technological approaches, including support in language workbenches. I highlight the benefits and limitations of each with regard to the challenges identified in previous chapters.

3.1 Introduction

In this chapter, we give a comprehensive study of the literature in light of the challenges identified in Chapter 1 and foundations presented in Chapter 2. Specifically, we look for tools and methods that increase flexibility in the manipulation of languages and their environment (**Challenge #1**) and support modularity and reuse in the development of DSLs (**Challenge #2**). We specifically look for methods that support reuse and composition of previously-defined (legacy) language artifacts that have not been designed specifically with reuse and composition in mind (e.g., with explicit interfaces and extension points). Language artifacts includes fragments or whole specifications of syntax and semantics, and services and tools supporting existing languages. When reusing and composing such legacy artifacts, various contraints and requirements must be satisfied. We introduce then in Section 3.2.

Figure 3.1 gives a graphical outline of the remainder of this chapter. To support the reuse of legacy language artifacts, fragments of syntax and semantics must first be imported into new language definitions, as depicted on the left side of Figure 3.1. We review current approaches to the reuse and assembly of legacy artifacts, along with the composition of languages, in Section 3.3. In Section 3.4, we then review interoperability capabilities in MDE (e.g., through the definition of generic tools and transformations) and flexible model manipulation (e.g., the ability to open and manipulate the same model in different modeling environments). Finally, in Section 3.5, we give

a comprehensive overview of the support for such facilities in popular and actively maintained language workbenches.

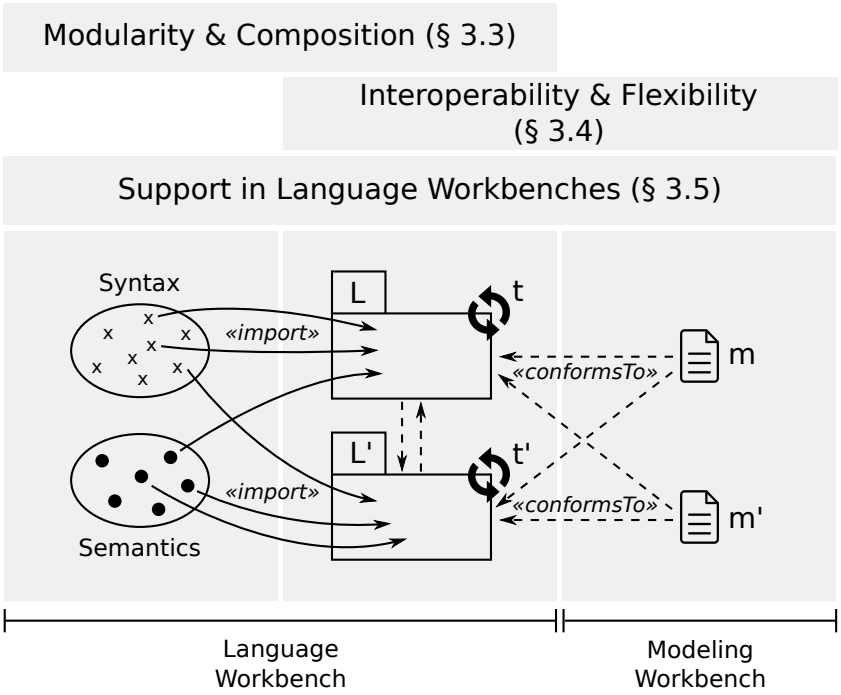


Figure 3.1: Graphical outline of this chapter

3.2 Constraints and Requirements

As mentioned in Chapter 2, the implementation of a language's abstract syntax consists of a set of interrelated types (*group of types*), implemented for instance as a set of classes automatically generated from a grammar or metamodel. The tools and services defined around a language, such as editors and transformations, use these groups of types to manipulate the conforming models (i.e., the sets of objects instantiating these classes). Achieving composition of languages, and reuse of tools from one language to the other thus implies to achieve composition and reuse over groups of types (also known as families of types [92]). The challenge we describe here is akin to Oliveira's "expression families problem", which formulates the problem of achieving reusability and composability when *families of interrelated types* are involved [200]. This problem is inspired from the original Expression Problem, which aims at defining new variants of a datatype and new functions over this datatype, without recompiling existing code and while retaining static type safety [270]. Translated to the context of SLE, variants correspond to language variants, and operations over these variants correspond to the tools and services supporting

a particular language, as illustrated in Figure 3.2. Language variants may have been generated one from the other (e.g., a language for hierarchical statecharts extending a language for flat statecharts), or correspond to variants found in the wild (e.g., the family of statecharts languages [63]). The goal is to achieve reusability and composability over the different variants and their tooling, so that tools defined for a particular language (e.g., a flattening transformation) can be reused for its variants. Constraints from the expression families problem still apply: reuse of tools and transformations over a family of languages should be checked statically, the semantics of type groups substitutability [92] should not be violated when creating variants or extensions of a given language, and separate compilation of variants and tools should be supported.

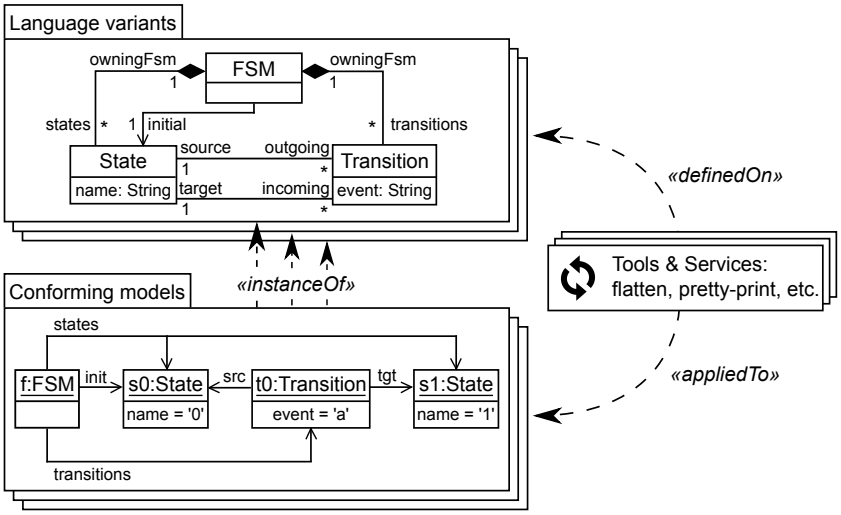


Figure 3.2: Reusing tools and services over language variants

From these observations and the challenges identified in Chapter 1, we identify the following list of requirements:

Composition at the specification and implementation levels When reusing existing languages to compose new ones, the composition must be done both at the specification level (i.e., composed specifications must be analyzable the same way original specifications are) and the implementation level (i.e., the implementations of composed languages must cooperate or be composed together). Moreover, the composition should ideally crosscut all language aspects: abstract syntax, concrete syntax, semantics, and tooling;

Fine-grained customization It is likely that existing languages do not exactly match the requirements of a new application domain. An ideal approach should support fine-grained customization of languages in order

to tailor them to specific contexts, and to implement syntactic and semantic variation points whenever needed. Typical customizations include restriction or extension of the expressiveness of an existing language, and refinement of its semantics;

Reuse of legacy artifacts Because we emphasize the reuse of existing artifacts, no assumption should be made on the way models, languages, and tools have been defined, for instance with modularization or reuse in mind. Tool support is a crucial facility for the adoption of a DSL. When composing languages, the tools and services (i.e., editors, analyzers, transformations) defined around them should be available on the resulting language, without having to rewrite or re-generate them;

Separate compilation When composing languages, their implementation should not have to be re-generated from scratch to form the resulting language. Instead, the composition mechanism should be able to reuse existing implementations as is (e.g., an interpreter), possibly with glue code to make the different parts fit together;

Tool-supported The proposed approaches for composition and interoperability should be tool-supported, i.e., provide concrete means for a language designer to apply them in practice. Ideally, these facilities should be integrated in a language workbench.

3.3 Modularity and Composition in DSL Engineering

There may be a significant overhead in creating both a language itself (syntax and semantics) and the infrastructure supporting it (e.g., an IDE with a dedicated set of tools and services). Numerous works propose to adapt the time-honored concepts of modularity, reusability, and composition to the specificities of software language development. For instance, some authors advocate the definition of reusable and composable *language modules* (also known as *language units* or *language fragments*) to support modular software language engineering [249, 197]. Others directly refer to the heritage of the component-based software engineering community and aim at applying the same techniques for modularizing language development [56], including reverse-engineering language modules from existing implementations [184]. In this section, we give a comprehensive overview of these works. As we shall see, the capabilities provided by a particular framework in terms of modularity and composition are highly dependent on the way languages are defined within this particular framework. Many design decisions influence the modularity and composition capabilities: the choice of an internal or external DSL, a grammar or metamodel to define the abstract syntax, a parsing-based approach

or a projectional editor, the implementation of the dynamic semantics in a high-level declarative formalism or directly in a programming language, the way tools and services are defined, etc.

Taxonomy of language composition In this section and the remainder of this thesis, we follow the taxonomy of Erdweg et al. in considering language composition as encompassing language *extension*,[1] *restriction*, *unification*, and *extension composition* [90].[2] The notion of *self-extension*, also introduced by Erdweg et al., refers to the ability of a language to extend itself, e.g., to embed specialized DSLs within it. We deem this notion out of the scope of this thesis, as it is not relevant for the development of *external* DSLs. Erdweg et al. consider that "language restriction does not require special support by language-development systems. Instead, a language restriction can be implemented as an extension of the validation phase of the base language: The extension rejects any program that uses restricted language constructs" [90]. This view of language restriction is also shared by other authors (e.g., Mernik [187]). Following this definition, the restricted language constructs are virtually hidden but are not removed from the language implementation. This definition of language restriction may be problematic in some cases. As an illustration, consider a scenario involving the extraction of a subset of the UML (e.g., the class diagram language) for reuse in another language. Simply "hiding" the unnecessary parts with validation rules to virtually extract the class diagram language is not satisfactory, as all the other constructs of UML are part of the resulting language (although not accessible). Language restriction should instead actually prune the unnecessary parts to avoid overloading the resulting language with unnecessary features and complicating its implementation. Thus, in this thesis, we prefer the following definition of language restriction, given in the style of Erdweg et al.'s paper [90]:

Definition 1 *A language-development system supports the restriction of a base language if the implementation of a base language can be pruned to extract a subset of interest. All other syntactic and semantic elements are removed from the base language to form the resulting language.*

3.3.1 Syntax Modularity and Composition

On the syntax side, modularity and composition capabilities for external DSLs development depend on the underlying techniques and formalisms: grammars,

[1] By analogy with inheritance in object-oriented programming, we use the terms *super-language* and *sub-language* to refer to the two languages involved in a language extension scenario.

[2] This notion of language composition contrasts with the notion of language coordination, which aims at integrating languages *a posteriori* for global reasoning or global simulation of a system described in different languages to support socio-technical coordination [46]

metamodels, or projectional editors. We review each of them in this section. According to the scope of this thesis, we pay particular attention to modularity and composition of metamodels.

Modularity and composition with grammars Regarding the grammar world, parsing-based techniques are known to have limited support for modularity and composition [81]. LR parsing, generalized parsing, scannerless parsing, PEG (Parsing Expression Grammars) parsing suffer from either ambiguity or overwhelming complexity when it comes to modularity and composition. The interested reader can refer to Diekmann and Tratt for a thoughtful discussion on the limitations of different parsing techniques [81]. Nonetheless, the field of grammar modularisation and composition is still under active development, as illustrated for instance by the work of Johnstone et al. who provide a module system for context-free grammars [149]. Naturally, these limitations have severe consequences on the frameworks for textual DSL development: for instance Xtext [93], which relies on the LL(*) parser generator ANTLR [217] internally, supports single grammar inheritance but does not support more advanced composition mechanisms, such as the composition of two arbitrary language grammars. As we shall see in Section 3.5, some textual language workbenches, such as MontiCore [168] and Spoofax [155], alleviate some of these limitations.

Modularity and composition with projectional editors In parallel to parsing-based editors, projectional editors (also known as Syntax Directed Editors – SDE) emerged in the eighties as an alternative way to envision program editing [245]. With projectional editors, users directly manipulate an AST conforming to the language, avoiding the need for parsing and setting aside the limitations of parsing-based techniques [262]. However, early attempts to implement SDE failed to get the acceptance of users, who were reluctant to the general feeling of projectional editors [158]. More recently, modern syntax-directed language workbenches (such as MPS [144] and the Intentional Domain Workbench [237]) alleviate most of these limitations [17]. Projectional editors are now considered again as a viable alternative to parsing-based editors [265]. Additionally, projectional editors open up new horizons for program editing: program text can be arbitrarily mixed with tables, symbols, and graphical elements. At the crossroads of parsing-based and syntax-directed editing, other authors attempt to combine the best of both worlds: the Eco editor, for instance, emulates the feeling of parsing-based editors while serializing files in their tree structure form [81].

Modularity and composition with metamodels When the abstract syntax of a language is defined by a metamodel, the problem of abstract syntax modularity and composition boils down to the problem of metamodel

modularity and composition. The modeling community has long recognized the need for reusing and composing all or parts of existing metamodels to create new ones, leveraging previously formalized domain knowledge [176]. Since metamodels are themselves models, model composition techniques also apply to metamodel composition. There is a rich literature of model and metamodel modularity and composition. These different approaches differ in their degree of automation, their support for fine-grained refinement, and the formalisms on which they can be applied [88]. We introduce a representative subset of them hereafter. The interested reader can refer to Clavreul's thesis for an in-depth analysis of model and metamodel composition techniques [55]. More formally, Hamiaz et al. use Coq [13] to formalize popular composition operators in MDE [121].

Within the Generic Modeling Environment (GME) [175], Lédeczi et al. use the UML class diagram as a metamodeling language and extend it with three operators dedicated to metamodel composition: the *union* operator which unifies two concepts, and the *implementation-inheritance* and *interface-inheritance* operators which refine the semantics of the UML inheritance relation for fine-grained specialization of concepts in composed metamodels [152, 176]. The MOF specification defines the *PackageMerge* operator, which is used to merge a *receiving package* and a *merged package* to form a new *resulting package* [204]. Informally, the *PackageMerge* operator operates recursively within two package definitions and unifies the common concepts (same name and meta-type) while copying the unique concepts from both side in the resulting package definition, so that "a resulting element will not be any less capable than it was prior to the merge" [204]. Emerson and Sztipanovits identify a set of common patterns that are often found in metamodels such as composition hierarchies, data-flow graphs, and state-transition patterns. They propose the *template instantiation* technique which consists in reifying these patterns as first-class entities to be instantiated on (i.e., merged with) concrete metamodels [88]. Fleurey et al. propose to compose models written in arbitrary languages using Kompose [102]. The Kompose approach operates on two phases: the language-specific *match* phase identifies common concepts in different models, and the language-agnostic *merge* phase integrates the common concepts to form the resulting model. Fleurey et al. illustrate their approach on the Ecore language, thus defining a composition operator for Ecore metamodels. Although they do not address the problem of semantics modularity and composition, all these works are good candidates for the definition of a composition operator for metamodel-based abstract syntaxes.

Along the UML and MOF specifications, the OMG defines the concept of *Profiles* that can be used to extend the syntax and – to some extent – the semantics of a given metamodel [207]. A profile consists of a set of *sterotypes*, *constraints*, and *tagged values*. Each stereotype extends a meta-class of an existing metamodel. The main intent of profiles is to tailor existing metamodels to particular domains or platforms. Profiles have been used extensively in

practice, especially to extend UML. Examples include the MARTE [206] profile for real-time embedded systems modeling, and the SysML [210] profile for systems engineering modeling. While profiles ensure that the tooling defined on the super-language can be reused, they have limited specialization powers: for instance, a profile cannot break the static semantics (i.e., constraints) of the metamodel it extends (e.g., the UML).

Clark et al. propose to leverage object-oriented practices in the development of languages and contribute to reusable and modular language design with a dedicated package specialization mechanism [51]. Amálio et al. present a theory of fragmentation for MDE revolving around a mathematical formalization of model fragments and ensuring that composed fragments yield valid models. Şutîi et al. present the multilevel metamodeling environment MetaMod. MetaMod consists of a minimalistic metamodeling language built around the concepts of *groups*, *fragment abstractions*, and *applications* with modularity in mind [242]. The interested reader can refer to the work of Atkinson and Kühne for a comprehensive study of language customization approaches in the MDE world [9]. The aim of these works is to propose a *prescriptive* way to engineer models and metamodels, and are not applicable for the modular reuse of *existing* languages. A promising approach however is the recent work of Bruneliere et al. who propose a dedicated DSL for customizing existing languages [36]. A limitation of their work is that it does not deal with the adaptation of all other language aspects when customization are applied (e.g., semantics and tooling).

3.3.2 Semantics Modularity and Composition

The need to modularize the semantics of programming languages suppose to modularize existing formal frameworks for semantics specification. This led for example to the definition of modular denotational semantics [41, 177] and modular structural operational semantics [195, 196]. Churchill et al. explicitly advocate the definition of reusable components of semantic specifications, named *funcons* (fundamental constructs), that are specified in I-MSOS [196] and can be independently developed and validated. The general idea is to reduce the development costs of new languages by translating their constructs to such funcons. The DynSem meta-language for dynamic semantics specification, for instance, is directly inspired from I-MSOS and, as such, provides similar modularity features [258]. Other frameworks for semantics specification, such as Redex [98] or K [227], inherit the modularity and composition features of rewrite systems.

At the implementation level, the operational semantics of a language is often implemented as an interpreter [110]. The problem of modular interpreter development thus garnered significant interest. In the functional programming community, one of the most popular class of solutions, originally proposed by Liang et al., uses monad transformers [178]. One of the main idiom used to

implement interpreters is the visitor pattern [110], which allows to separate ASTs from the algorithms that operate on them. The original visitor pattern has however poor support for composition and extensibility. Several authors proposed extensions to the visitor pattern to alleviate these limitations. The interested reader can refer to the work of Oliveira for a comprehensive review of these extensions [200].

Object algebras were originally proposed by Oliveira and Cook as a solution to the expression problem that does not require advanced typing features such as F-bounded quantification and can thus be implemented in most object-oriented programming languages [201]. Several authors proposed to leverage object algebras for implementing languages in a modular way. Gouseti et al. identify that object algebras support the implementation of recursive data types (such as ASTs) and as such support modular definition and extension of both the syntax of a language and the algorithms operating on it (e.g., interpreters, static analyses) [116]. More recently, Inostroza and van der Storm propose a new method for the development of modular interpreters relying on the theory of object Algebras [201]. In their approach, interpreters exposing heterogeneous signatures can be composed and extended thanks to *implicit context propagation* [142]. As an illustration, they demonstrate how to implement a modular interpreter for Featherweight Java [141] as an assembly of language fragments, and how to extend the resulting language to add support for state. All these works propose specific patterns for the definition of languages that offer better support for modularity and extensibility. They are however not immediately applicable to languages that have not been defined with modularity and reuse in mind.

The problem of language composition also arises when engineering software systems written in different general-purpose languages (say, Python and C#). In this case, the goal is not to create a new unified language *per se*, but to make programs written in different languages cooperate. A well-known approach is to leverage Foreign Function Interfaces (FFI) and inter-process communication (IPC) mechanisms, but these mechanisms are clumsy to manipulate and not reified at the language level. It is clearly easier to compose interpreters that run on the same underlying virtual machine: for instance, interpreters for JVM-based languages such as Groovy, Java, and Scala. Conversely, it is much more challenging to compose interpreters that run on heterogeneous platforms. The challenge here is to "glue" together two interpreters, by aligning the basic data types they manipulate and coordinate their execution. Barrett et al.'s approach aims at gluing together GPLs such as Python and PHP using meta-tracing techniques [14, 15]. Although in their approach the languages are not composed *per se*, the techniques they propose may be relevant for coordinating the execution of interpreters when several languages are composed.

Finally, although these techniques are beyond our scope, it is interesting to note that several authors demonstrated the possibility to create language units using attribute grammars [154, 229, 188]. As attribute grammars unify

the definition of abstract syntax, concrete syntax, and semantics, these works support, to some extent, modularity and composition of all aspects of a language. However, because of this unification, all the constituents of a language are mixed in the same artifact. In our opinion, this hampers separation of concerns in language development and independent evolution of each concern.

3.4 Interoperability and Flexibility in MDE

Software languages are constantly evolving to meet new requirements. This is especially true for DSLs: by their very nature, they must evolve with the domain they abstract. In the MDE technological space, these changes mainly impact the metamodel of a language. They may result from hand-made customizations or systematic applications of composition mechanisms such as the ones presented in Section 3.3. Because models, metamodels, and transformations are highly interrelated, is it often hard to evolve one artifact without evolving the others. This lack of flexibility in MDE has been identified by several authors who argue that the problem lies in the overconstraining dependency of MDE artifacts towards metamodels through the conformance relation [109, 165]. As a result, evolving a language has severe consequences on the models conforming to it and the ecosystem defined around it [214].

A modeling environment dedicated to a particular DSL – including editors, interpreters, services, etc. – revolve around a particular metamodel, and can only be used to manipulate models conforming to this particular metamodel. When the metamodel evolves, all depending artifacts must evolve to cope with new or evolving concepts and constraints. This raises many questions regarding the support of MDE for evolution and interoperability of environments. One way to tackle this problem is to support the automatic co-evolution of metamodels and the artifacts that depend on them (e.g., models and transformations). Numerous work try to address these issues, including model-metamodel co-evolution (e.g., [268, 118, 129, 131, 132, 198, 130, 111, 128, 80]) and transformation-metamodel co-evolution (e.g., [49, 173, 183]). Other authors propose to relax the conformance relation that stands between models and their metamodels to increase flexibility in the manipulation of models [278, 165]. We do not discuss further the limits of the conformance relation here. Chapter 5 gives a thorough analysis of the limits of the conformance relation from theoretical and experimental points of view. For higher-level discussions of the notions of evolution and compatibility in MDE, the interested reader can refer to the work of Kühne [171] and Meyers and Vangheluwe [193].

In this thesis, we name *flexible modeling* the ability for a language user to open and manipulate its models in various modeling environments defined around various languages. As a concrete example, consider the case of statecharts languages. Many statechart languages are developed within independent company for specific purposes. Language users must be given the flexibility to

open their statechart models in different statechart modeling environments, even if their models do not conform to the particular statechart language of this environment. Thus, they can reuse the tools and transformations defined by other engineers, and foster model sharing between stakeholders. To this end, we review in Section 3.4.1 approaches that support the definition of generic transformations and the reuse of transformations.

3.4.1 Model Transformation Reuse

In this thesis, we consider every tool and service defined around a given language to be a model transformation manipulating the models conforming to it: editors create and modify them, analyzers check them, interpreters and compilers execute and generate code from them, etc. It follows that increasing the compatibility and interoperability of modeling environments implies to tackle the problem of transformation reuse and genericity. In this section, we examine state of the art approaches to the definition of reusable and generic model transformations that alleviate the limitations of the overly restrictive conformance relation, and provide more flexibility in the manipulation of models. Several approaches have been proposed over the last decade for model transformation reuse. These approaches can be divided into two categories: model transformation reuse without adaptation (i.e., reuse between isomorphic metamodels) and reuse allowing to fix structural heterogeneities through adaptation.

Reuse without Adaptation Model transformation reuse without adaptation was first proposed by Varró and Pataricza who introduced *variable entities* in patterns for declarative transformation rules [257]. These entities only express the concepts (types, attributes, etc.) required to apply a transformation rule. This allows tokens with these concepts to match the pattern and be processed by the rule. Later, Cuccuru et al. introduced the notion of semantic variation points in metamodels [67]. Variation points are specified through abstract classes defining a *template*. Metamodels can fix these variation points by binding them to classes extending the abstract classes.

Steel and Jézéquel propose to leverage existing works in object-oriented typing to better cater a model-oriented context. They introduce the concept of model type to define structural contracts over metamodels [239]. The idea is to supersede the conformance relation between models and metamodels with a typing relation between models and model types. Guy et al. build on Steel and Jézéquel's model types and describe a family of model-oriented type systems that differ in their support for adaptation (isomorphic and non-isomorphic subtyping) and partial or total subtyping. One benefit of model types is that they are expressed in the same formalism as metamodels, which means that they can be directly manipulated using classical tools of the modeling ecosystem, such as model transformations, editors, etc. Sun

et al. propose to enrich model types with static semantics rules, to go beyond pure structural substitutability and support behavioral substitutability [241]. Sánchez Cuadrado and García Molina propose a notion of substitutability based on model typing and model type matching [228]. Instead of using an automatic algorithm to check the matching between two model types, they propose a DSL to manually declare the matching.

All these approaches require either to insert new concepts in the definition of languages or transformations, or to manually declare matchings between two languages, even when they are structurally similar. Because these approaches are intrusive, they are hardly applicable to the reuse of legacy tools and transformations that cannot be modified.

Reuse through Adaptation Adaptation enables the reuse of model transformations between metamodels in spite of structural heterogeneities. Two kinds of approaches exist. The first one adapts models conforming to a metamodel MM into models conforming to a metamodel MM' on which is defined the transformation of interest. The second one adapts a transformation defined on MM' to obtain a valid transformation on MM, using a higher-order transformation.

De Lara and Guerra present the notion of *concept*, along with *model templates* and *mixin layers* [70, 225]. These notions are borrowed from the idea of generic programming, as found in mainstream programming languages (e.g., templates in C++ and Java). *Concepts* are similar to model types as they define the requirements that a metamodel must fulfill for its models to be processed by a transformation. However, in their current form, *concepts* do not benefit from subtyping relations between different concepts, and a metamodel must be explicitly binded to the *concept* of each transformation to be reused. The authors also propose a DSL to bind a metamodel to a *concept* and a mechanism to generate a specific transformation from the binding and the generic transformation defined on the *concept*. Cuadrado et al. extend this binding mechanism to go further than strict structural mapping by renaming, mapping, and filtering metamodel elements [65]. *Concepts* are inspired from parametric polymorphism (aka. generics) while model types are inspired from inclusion polymorphism (aka. subtype polymorphism). Parametric and inclusion polymorphism serve similar purposes and are known to be complementary [40].

De Lara et al. introduce *a-posteriori typing* for MDE, which allows to uncouple the creation type of a model from its classification types and to reclassify models after their creation to enable their manipulation in other contexts [73]. A-posteriori typing goes beyond previous approaches by enabling instance-level classification, which inherits the benefits and drawbacks of dynamic typing, and is implemented in the MetaDepth platform [69].

Kerboeuf and Babau present an adaptation DSL named *Modif* which handles deletion of elements from a model (that conforms to a metamodel MM) to make it substitutable to an instance of the metamodel MM' [157]. For this, a trace of the adaptation is saved to be able to go back from the result of the transformation (conforming to MM') to the corresponding instance of MM. Garcia and Díaz proposed to semi-automatically adapt a transformation with respect to metamodel changes [112]. A classification of metamodel evolutions is proposed as well as automatic adaptations of the transformation for some of them.

3.5 Support in Language Workbenches

As introduced in Chapter 2, language workbenches are one of the main innovation medium in software language engineering. Most language workbenches support the definition of the abstract syntax, the concrete syntax, the semantics, and the tooling of DSLs [91]. As such, they have to deal with modularity and composition at all levels of language definition coherently. In this section, we present the mechanisms provided by actively maintained language workbenches for modularity and composition (Section 3.5.1), and their support for flexible manipulation of models and programs (Section 3.5.2).

Since this thesis focuses on the engineering of external DSLs, we do not review frameworks dedicated to the engineering of internal DSLs (e.g., using meta-programming, "languages as libraries", or dialects) such as Cedalion [179], LMS [224], Racket [101], Recaf [25], SugarJ [89], TSL [202], or XPL [50]. As a complement to the state of the art presented here, the interested reader can refer to the recent study of Erdweg et al. for complementary information on the landscape of language workbenches features [91], or to the website of the language workbench challenge which is an initiative of the community to regularly evaluate and compare various language workbenches on challenging case studies.[3]

3.5.1 Modularity and Composition

Monticore MontiCore is a language workbench for the development of textual domain-specific languages [167]. MontiCore relies on an enriched BNF-like grammar formalism that defines the abstract and concrete syntax of languages. Their semantics can be defined either denotationally or operationally, based on automatically-generated visitors. MontiCore provides two mechanisms for language composition: *language inheritance* – used to extend an existing language while refining its grammar productions – and *language embedding* – used to complete intentional holes in the specification of a language with other language specifications [168, 120]. In both cases, language designers

[3] http://www.languageworkbenches.net/

must introduce explicit *interfaces* in language definitions that serve as "hooks" for inheritance and embedding. MontiCore supports *multiple* inheritance of grammars, which can be used in practice to unify two independent languages. Recently, the same authors proposed a way to also compose the generated visitors when languages are composed [125]. Each nonterminal in a MontiCore grammar results in a concrete class at the implementation level. When composition operators are used, new classes are created by inheriting from their corresponding classes in the super-languages. In some cases, this may break the consistency of type groups and result in unsafe manipulations of the resulting ASTs, as explained by Steel and Jézéquel [239].

LISA LISA is a language workbench that relies on attribute grammars for the definition of textual domain-specific languages [190]. LISA supports multiple attribute grammars inheritance [189], which serves as a mean for incremental language development, language extension, restriction, and unification [187]. Grammars can be inherited and subsequently refined by overriding the production rules and the associated computations. From attribute grammars specifications, LISA automatically derives a set of language-specific tools such as editors, visualizers, and evaluators [127]. One issue with the approach is that all language concerns (abstract and concrete syntax, semantics) are mixed in a single meta-language: the attribute grammar formalism of LISA. As a result, it is hard for a language engineer to evolve each concern independently, e.g., for engineering syntactic and semantic variation points. Also, the authors do not directly address the problem of reusing tools and programs across versions or variants of a language.

Spoofax Spoofax [155, 269] is a language workbench dedicated to the engineering of textual DSLs. The main idea of Spoofax is to foster the use of high-level and declarative meta-languages to specify each aspect of a language: SDF for syntax definition, NaBL for name binding rules, TS for type analysis, Stratego and DynSem for dynamic semantics, etc. [259]. A language designer can directly conduct analyses at the specification level, for instance to check for correctness properties and generate documentation. The combination of these high-level specifications forms the overall language specification, from which concrete implementations (e.g., interpreters), proof materials (e.g., Coq specifications) and IDE services are derived. Semantics specification in Spoofax relies on the Stratego [33] program transformation language.[4] Stratego specification consists of term rewriting rules that are used for the analysis, manipulation, and generation of programs. Concretely, language designers can derive interpreters, compilers, or static analyses from Stratego specifications. Language extension and composition in Spoofax is supported thanks to the composability of its

[4]The DynSem meta-language [258] is currently under development as an alternative to Stratego.

underlying meta-languages SDF and Stratego. However, Völter and Visser recognize that "such extensions requires anticipation of extensibility in the design of the base language by including proper extension points" [266].

JetBrains MPS The Meta-Programming System (MPS) is a language workbench developed by JetBrains and based on projectional editing [144]. MPS has notably been employed to re-engineer programming languages such as C (through the mbeddr environment [263]) and Java, and is also used for engineering DSLs. MPS offer dedicated meta-languages for specifying, for instance, the syntax, projection rules, type system, and generator of a DSL. Because MPS relies on projectional editing, it does not suffer from the problem associated to parsing-based techniques and proposes advanced extension and composition mechanisms [262]. However, Voelter admits that "in many cases, languages have to be designed explicitly for reuse, in order to make them reusable" [260]. Similar problems arise when considering language composition. Also, MPS suffers from similar problems as we have described for MontiCore. In particular, the extension of concepts in a sub-language or composed language relies on classic object-oriented inheritance *in the small* (i.e., between individual concepts), and not *in the large* (i.e., between the whole groups of types defining a language's concepts), which can lead to type groups inconsistency and unsafe manipulations [239].

Neverlang The idea of feature-oriented language development is to support the definition of languages *à la carte*, as an assembly of independently-developed language components [170]. One proponent of this approach is the Neverlang framework, which proposes the concept of *slices* and *modules* that materialize a language construct along with its semantics [249]. These slices can then be assembled as a puzzle to form new languages. This idea has for instance been applied to implement a "growable" programming language for teaching purposes [42]. Continuing in the same direction, Cazzola and Vacchi later proposed to leverage Scala's traits as an alternative implementation technique for modular development of external DSLs [43]. Their approach addresses the modularization of both the syntax and semantics of external DSLs, hence supporting the implementation of DSL as an assembly of components and maximizing code reuse between languages. It is however not clear how their approach ensures type groups consistency, and how they address the reuse of tools from one language to the other.

Xtext Xtext is a framework for engineering textual DSLs [93] that internally relies on EMF and the LL(*) parser generator ANTLR [217]. The definition of a language in Xtext consists of a BNF-like grammar specification. From this specification, Xtext derives a set of integrated tools such as a parser, a full-blown editor, etc. Xtext grammars can inherit one another to reuse

and refine production rules. Due to the limitations of the underlying parsing technology, Xtext does not support more advanced language composition scenarios such as modular and incremental language definition, and arbitrary grammar composition.

MetaEdit+ MetaEdit+ is a mature environment dedicated to the engineering of domain-specific modeling languages [247, 248]. In Metaedit+, languages are specified by a metamodel upon which language designers can define static semantics rules, graphical representations, and code generators. While authors report that MetaEdit+ supports certain kinds of language extension and composition [91], we were not able to find any documentation or publication describing these mechanisms in detail.

Rascal Rascal [161] is the successor of the ASF+SDF meta-environment [251]. It is a full-blown environment for source code analysis and transformation, and is also used as a language workbench [253]. Concrete syntaxes in Rascal are defined using grammars that can be arbitrarily composed at the price of the use of generalized parsing technologies, which may be possibly ambiguous as stated in Section 3.3. The corresponding abstract syntaxes are implemented as algebraic data types [253]. From these specification, Rascal supports the generation of domain-specific editors, outline, etc. as Eclipse plug-ins.

3.5.2 Flexible Model Manipulation

Support for flexibility in the manipulation of models depends on the choice of grammars or metamodels for syntax specification.

In the grammar world, using parsing-based techniques, models and programs are serialized and persisted in their concrete syntax form. From a given file, there is no way to statically infer (i.e., without parsing the file), the language that was used to create and serialize it. The only information one can extract from a model serialized in its concrete form is the file type and extension, which is not precise enough to determine its exact language. As a concrete example, it is not possible to determine *a priori* which particular version of the C++ language was used to write a given C++ file. The only way to find out is to actually try to parse the file using a compiler implementing a given version of the C++ standard. If it encounters a construct it cannot process, it may fail unpredictably and raise an error. Conversely, parsers provide some flexibility as they always allow to try to parse a model. Parsing a program written in the C++14 standard using a C++03 compiler may succeed, or may fail if it contains a construct specific to C++14. All in all, this amounts to some kind of "Russian roulette": the flexibility in model loading occurs at the price of safety.

In the metamodel world, models are serialized in their abstract syntax form (typically, as an XML file conforming to the appropriate metamodel

schema). This is for example the case in EMF, which serialize each model in the XMI format with an explicit URI identifying the metamodel it conforms to [240]. Thus, the XML parser can determine *a priori* which metamodel must be used to parse a model, just by looking at its URI. This means that, in this case, parsing a model is always safe. Conversely, this mechanism hinders flexibility in model manipulation: it is not possible to bypass the conformance relation, and a model can only be manipulated using its exact metamodel. If the metamodel evolves (and, consequently, it's URI), the conforming models cannot be loaded anymore. The safety in model loading occurs at the price of flexibility. As mentioned in Section 3.4.1, recent works in MDE propose to discard the conformance relation and to classify models *a posteriori* [72]. These approaches are in essence similar to dynamic typing in programming languages and provide similar benefits and drawbacks. In particular, they do not allow to *statically* state whether a given model can be manipulated through a given metamodel.

As a summary, the MDE approach to model manipulation is always safe, but hinder the flexible manipulation of models in different environments. On the other side, the grammarware approach offers greater flexibility by always allowing to "try" to load a model or program, but is not safe as the parser may unpredictably fail if it encounters a construct that is not supported. A promising idea would be to take the best of both worlds: the safety of the MDE approach, and the flexibility of the grammar approach. In this vision, it should be possible to statically list all the languages that can be used to load and manipulate a given model, and to safely apply the corresponding tools and services upon it.

3.6 Summary

The vast majority of the approaches we introduced in this chapter prescribe a development method for modular and reusable development of software languages. They suppose the explicit definition of interfaces and extension points in the base languages, or the application of specific development patterns. This means that modularity and composition must be *anticipated*, which hampers the applicability of these approaches on legacy artifacts that have not been defined in a particular way. Similarly, approaches for generic transformations and flexibility in MDE suppose the adaptation of either the models, the languages, or the transformations themselves. An approach for modular development of DSLs, reuse of tools and transformations, and flexibility in model manipulation that is applicable to legacy languages and tools is still to be developed.

Regarding operators for language composition, language restriction is an essential facility when reusing language definitions whose scopes are too broad, such as UML. Current approaches to language restriction are however not satisfactory as they only hide the unwanted elements without pruning them.

Also, the customization capabilities of the syntax and semantics of reused languages differ from one approach to the other. In the case of legacy language composition, fine-grained composition and customization mechanisms are however needed.

In the next chapters, we present new contributions that address these limitations by providing modularity, reuse, and interoperability for legacy languages in a seamless way.

Part II

Contributions

Chapter 4

On Language Interfaces

In this chapter, I develop our first contribution: the notion of language interface*. After introducing the general idea of language interfaces in Section 4.1, I summarize in Section 4.2 the use and benefits of interfaces in software engineering and programming. Then, I outline in Section 4.3 the benefits of language interfaces in the context of software language engineering.*[1]

4.1 Introduction

The implementation of a DSL highly depends on the mechanisms provided by the language workbench. Consequently, the various services associated to a DSL (such as editors, model checkers, debuggers, and composition operators) are tightly coupled to the DSL's implementation. Due to the lack of explicit abstraction mechanism on top of DSLs, the definition of services requires in-depth understanding of the DSL implementation (for instance, metamodels for abstract syntax definition and OCL for the description of static semantics). The information a service requires on a DSL is scattered over various artifacts, and expressed in different formalisms. It is thus complex for language designers to gather the appropriate information required to implement a particular service. Moreover, this high coupling of the services with the DSL implementation prevents their reuse from one DSL to the other, even when they share commonalities.

While the implementation of most language services crosscut the different DSL concerns (abstract syntax, concrete syntax and semantics), they usually require only partial information on each, possibly given in a new form which abstract the initial concrete language implementation (e.g., control-flow graph). The lack of possible abstraction prevents the reuse of services among close DSLs, and increases the complexity of service implementation.

Leveraging the time-honored concept of *interfaces* from software engineering to software language engineering can facilitate the reuse of language services

[1] The material presented in this chapter led to the following publication: [79].

among various DSLs, and reduce the required language implementation expertise for their development. Language interfaces enable to encapsulate language (and model) functionalities in an accessible fashion, can be used to guide and validate the correct usage of languages with respect to specific purposes, and even facilitate substitution of specific languages for different services. Language interfaces can also define the protocol one should follow to interact with its models to enable concurrent execution and global analysis of heterogenous models. Recently, different initiatives such as the Language Server Protocol[2] or the concept of micro-grammars [34] proposed to make explicit the concept of language interface to ease the definition and reuse of services. We hope our conceptualization of language interfaces will serve as an underlying theory for the development of such approaches.

In this chapter, we review the various uses and benefits of *software interfaces* in software engineering, including in programming languages and component-based software engineering. We then demonstrate the alignment of the benefits of software interfaces with regards to current challenges in software language engineering, and shape the concept of *language interface*. Our main intent in this chapter is to reflect on past experiences and propose a vision for software language engineering. This chapter is thus purposely abstract, and does not aim at giving a fixed definition of language interfaces. We refine and concretize our vision in Chapters 5 and 6 with a special kind of structural language interface: the model type.

4.2 Interfaces in Software Engineering

The time-honored concept of interface has been studied since the early days of computer science in many areas of programming and software engineering. Despite variability in their exact realization, interfaces invariably rely on common fundamental concepts and provide similar benefits. Historically, the notion of interface is intrinsically linked to the need for *abstraction*, one of the fundamental concept of computer science. As stated by Parnas et al., "an abstraction is a concept that can have more than one possible realization" and "by solving a problem in terms of the abstraction one can solve many problems at once" [216]. One can *refer* to an abstraction, leaving out the details of a concrete realization and the details that differ from one realization to another [156]. Originally, in programming, the key idea was to encapsulate the parts of a program that are more prone to change into so-called modules, and to design a more stable *interface* around these change-prone parts. This concept, known as *information hiding*, eliminates hard-wired dependencies between change-prone regions, thereby protecting other modules of the program from unexpected evolution [215].

[2] https://github.com/Microsoft/language-server-protocol

As different realizations may be used in place of an abstraction, interfaces also foster reuse: one module can substitute another one in a given context provided that they realize the same interface expected by a *client*. The choice of a concrete module is transparent from the point of view of the client of the interface. Because interfaces expose only a portion or an aspect of a realization, leaving out some details, the nature of an interface is highly dependent on the nature of the concrete realization it abstracts from. Following the evolution of programming paradigms, authors have thus defined various kinds of interfaces for various concrete realizations: modules (*module interfaces* in Modula-2 [276]), packages (*package specifications* in Ada [139]), objects (*protocols* in Smalltalk [115]), or components in Component-Based Software Engineering (CBSE) [126], to name a few.

The expressiveness in which one can specify an interface for a given kind of realization also varies: interfaces over objects in standard Java consists merely of a set of method signatures, while languages supporting design-by-contract enable the expression of behavioral specifications, e.g., in the form of pre- and post-conditions on those signatures [192]. The expressiveness of contracts themselves range from purely syntactical levels to extra-functional (e.g., quality of service) levels [20]. Interfaces are also closely linked to the notion of *data type* in programming languages [38]. Types abstract over the concrete values or objects manipulated by a program along its execution. Type systems use these types to check their compatibility, reduce the possibilities of bugs, and foster reuse and genericity through polymorphism and substitutability [40].

While in programming languages the realizations hidden behind a given kind of interface are mostly homogeneous, this is usually not the case in CBSE. As an illustration, component models enable communication between components written in different programming languages and deployed on heterogeneous platforms [236]. In such a case, interfaces abstract away from implementation technologies to enable interoperability between heterogeneous environments. The associated runtime model is most of the time unaware of the functional aspect of components and uses generic interfaces to manage their life cycle (e.g., deploying or updating the configuration of a component). Most component models also provide the notion of required interface as a mean to make explicit the dependencies between components and to reason about how different components interact together and must be composed.

In summary, interfaces are used in many ways and vary according to the purpose they serve. While "interfaces as types" mainly target the safe reuse and substitutability of modules and objects in different contexts and focus on functional aspects, the interfaces used in CBSE allow components to be independently developed and validated, and focus on extra-functional aspects. From these observations, we explore in the next section possible applications of the concept of interfaces at the language level, to improve the current practice of software language development.

4.3 Interfaces for Software Language Engineering

"Software languages are software too" [96] and, consequently, they inherit all the complexity of software development in terms of maintenance, evolution, user experience, etc. Not only do languages require traditional software development skills, but they also require specialized knowledge for conducting the development of complex artifacts such as grammars, metamodels, interpreters, code generators, model transformations, or type systems. As stated in Chapter 2, the need for proper tools and methods in the development of software languages recently led to the emergence of the software language engineering research field. While the notions presented in this chapter are applicable to both general-purpose languages and domain-specific languages, we put a particular emphasis on the specificities of DSLs.

New DSLs are usually developed using a language workbench, a "one-stop shop" for the definition of languages and their environments [106]. The main intent of language workbenches is to provide a unified environment to assist both language designers and users in, respectively, creating new DSLs and using these. Modern language workbenches typically offer a set of meta-languages that language designers use to express each of the implementation concerns of a DSL, along with tools and methods for manipulating their specifications [259].

One of the current trends in SLE is to consider more and more languages as first-class entities that can be extended, composed, and manipulated as a whole. However, to the best of our knowledge, there exists no previous work dealing with the explication of *language interfaces*, i.e., interfaces *at the language level*, explicitly separated from language implementations. The main purpose of language interfaces is to provide the appropriate abstraction to ease the manipulation of languages as first-class entities and foster reuse in SLE. To motivate the need for language interfaces, we explore in this section some of the current challenges faced in SLE and highlight how they match the challenges that have been addressed with the use of interfaces in programming and software engineering.

4.3.1 Supporting the Definition and Reuse of Services

The development of services (e.g., code and documentation generators, static analyses) is an essential part of the development of software languages. Defining a new service requires gathering the appropriate information which is often scattered in the various concerns of a given language (abstract syntax, concrete syntax, semantics), in different formalisms. Naturally, the information to be extracted varies according to the *purpose* of the service. Some services, such as simple static analyses, only require access to a subset of the syntax definition of a language. More complex services may require to aggregate information from different sources, or an abstraction thereof. Moreover, services may be defined at the meta-language level and used at the language level (e.g., analysis of the

completeness of a formal semantics specification), or at the language level and used at the model level (e.g., dead code elimination for programs written in a particular language). Rather than searching for the right information in the various concerns of a language, one can first design a language interface that aggregates the appropriate information in an easily manipulable form for a specific purpose. Using the appropriate interfaces eases the cognitive effort and abbreviates the definition of services. Moreover, since different languages can match the same interface, the services written on an interface can be applied to all compatible languages. Some generic services do not even require any information on the syntax or semantics of a language. Generic debuggers, such as the one found in the GEMOC studio [30] or on top of the ASF+SDF meta-environment [252], only require the ability to start, pause, or inspect the execution of a model or program [30]. Such common operations can be captured in a generic interface, and any language implementing it can benefit from debugging facilities.

4.3.2 Supporting the Coordination of DSLs

In the last decade, the development of MDE and the advances in language workbenches have strengthened the proliferation of DSLs. MDE advocates the use of DSLs to address each concern separately in the development of complex systems with appropriate abstractions and tools [108]. As a result, the development of modern software-intensive systems often involves the use of multiple models expressed in different DSLs to capture different system aspects [60]. This trend is very similar to what was proposed by Ward in his seminar work on language-oriented programming [271]. Even in traditional software development, multiple languages are often used to describe different aspects of the system of interest. Java projects, for instance, typically consist of Java source files, XML files describing the structure of modules and the deployment scenarios, Gradle build files expressed in Groovy, and various other artifacts for building and deploying.

When multiple languages are used, the need for relating their "sentences" (that, for example, describe the same underlying system in different languages) arises. In this context, models are seldom manipulated independently: checking a given property on a system requires gathering information that is scattered in various models written by various stakeholders in various languages. In addition, the set of languages used to describe a given system is likely to change over time. A new, more expressive language can replace an existing one. New languages may be added, merged, or split. While the support of language workbenches for engineering isolated languages is becoming more and more mature, there is still little support for relating concepts expressed in different DSLs together. The necessity of coordinated use of languages used in the development of a given system has recently been recognized [60, 46]. One promising approach is to leverage language interfaces to expose the appropriate

information that allows relating concepts from one language to the other [53]. The type systems of two languages, for instance, may be related through the appropriate interface that would expose their respective type definitions to allow their integration. Overall, the challenges that must be tackled are very similar to the ones that were faced a few decades ago with the use of modules in software development.

The coordinated use of DSLs engineered with different language workbenches is even more challenging. Indeed, there are no abstraction mechanisms, at the language level that allow abstracting from the concrete implementation techniques of a language workbench, e.g., a particular meta-language for defining the abstract syntax or a particular implementation technique for the semantics. The use of interfaces and connectors has been thoroughly studied in the context of CBSE to alleviate the problem of technological space compatibility. Explicitly separating the implementation of a language from its interface can help to break the barriers between language workbenches if a common agreement on technology-agnostic interfaces is found.

4.3.3 Enabling Language Composition

We project the CBSE vision of reusable building blocks to software language engineering. The idea of building reusable and independently-validated language components to ease the definition of new languages has already been studied by several authors. A language component, such as a simple expression language or a for-loop construct, can be defined and thoroughly validated independently and then reused as such in other languages that encompass the definition of expressions or for-loops. In the same way languages are defined, the definition of such component encompasses both the definition of its abstract syntax, concrete syntax, and semantics. This leads to what is known as compositional language engineering: the ability to design new languages as assemblies of existing language components, thus lowering the development costs [168]. In this context, language interfaces serve as support to denote what is provided and required by each component. They are later used to reason about the composition of several language components and the correctness of the assembly (e.g., through interface compatibility checking).

Interfaces are used to detect whether the constructs of different language components are in conflict, without having to dive into their intrinsic implementation details. Language interfaces are also the concrete mean for several language components to communicate with each other at runtime. For instance, when two executable languages are composed together, their interpreter must also be coordinated. Language interfaces provide the appropriate information for relating the two interpreters.

4.3.4 Engineering Language Families

A language family (also known as language product line) is a set of languages that share meaningful commonalities but differ on some aspects [169, 185]. Finite-state machine languages, for instance, are all used to model some form of computation but expose syntactic variation points (e.g., nested states, orthogonal regions) and semantic variation points (e.g., inner or outer transition priority) [63]. Altogether, these different variants form a family of finite-state machine languages. Similarly, when a language evolves, the subsequent versions form a set of variants, which raises the question of backward and forward compatibility between them [171]. Recently, different approaches have been proposed for automating the generation of such language variants based on earlier work from the software product line engineering community [170, 250].

In the presence of a language family, language designers must be given the possibility to reuse as much as possible the environment and services (e.g., editors, generators, checkers) from one variant to the other. Naturally, the opportunities of reuse must be framed, as not all services are compatible with all variants. Language interfaces can be employed to reason about the commonalities of various language variants and the applicability of a given service or environment. Model types [239], for instance, can be used to assign a type to a language and specify the safe substitutions between different artifacts based on subtyping relations [119].

The definition of those types is precisely an example of language interface. Types abstract over the details of the implementation of different variants and are used to reason about substitutability between them. In this context, language interfaces can support the definition of common abstractions that are shared by all or by a subset of the members of a family, abstracting from the details that vary from one member of the family to the other.

4.4 Conclusion

The lack of abstraction in the manipulation of DSL implementations complexifies the definition of services over DSLs (e.g., debuggers, generators, composition operators) and hampers their reuse. In this chapter, we have reflected on the use of interfaces in programming and software engineering, and advocated the definition of interfaces at the language level for software language engineering. We have shown that various challenges of today's language development can be addressed through the use of language interfaces. The concept of language interface presented here is purposely abstract. The two next chapters concretize the idea of language interfaces for two specific purposes: interoperability and flexibility in MDE (Chapter 5), and modular DSL development (Chapter 6). To this end, they focus on one particular kind of structural language interface: model types.

Chapter 5

Safe Model Polymorphism for Flexible Modeling

In this chapter, I present our second contribution: an approach for manipulating models, in a polymorphic way, through various language interfaces. This mechanism, named model polymorphism*, increases compatibility and interoperability of DSL environments, and offers flexibility in the manipulation of models to DSL users. In Section 5.1, I introduce the context and motivation of our contribution. In Section 5.2, I analyze the fundamental properties of the conformance relation in MDE, and list its limitations from theoretical and experimental points of view. In Section 5.3, I present our proposal on superseding the conformance relation with a typing relation enabling greater flexibility in the manipulation of models. I detail in Section 5.4 its implementation in the Melange language workbench, and evaluate our contributions in Section 5.5. Finally, I conclude in Section 5.6 on the observed benefits of our contribution.*[1]

5.1 Introduction

The constant evolution of DSLs and their necessary coordination raise the need of more flexibility for DSL users in their manipulation of the models through various modeling tools (either using different versions of a given DSL or different DSLs). This requires increased compatibility of modeling tools between different versions of a given DSL, and interoperability between the modeling tools used by different stakeholders.

MDE technologies assist domain experts in defining problem-space DSLs, without requiring strong skills in language implementation or compiler construction [137]. Using MDE technologies, new DSLs are typically first specified through metamodels that define their abstract syntax. The MDE community has developed a rich ecosystem of interoperable, generative tools defined over

[1] The material presented in this chapter led to the following publication: [78].

standardized object-oriented metamodeling technologies such as EMOF [204]. These tools can generate supporting DSL tools such as parsers, code generators, and other integrated development environment services.

MDE technologies strongly rely on the *conformance relation*, which states that a model conforms to a metamodel if each of its elements is an instance of a meta-class defined in the metamodel. As a result, *a model conforms to a unique metamodel*: the one used to create it. This implementation-oriented view prevents *model polymorphism*, i.e., the ability to manipulate a given model through different interfaces, each one associated to a particular DSL for a specific domain of expertise or particular modeling tools. Therefore, while models could be manipulated by a family of similar DSLs, in practice it is not possible.

In the last decade, significant efforts were devoted to model and metamodel co-evolution, as presented in Chapter 2. However, we demonstrate in this chapter that an important part of the required interoperability and compatibility appears between structurally similar DSLs, opening up the possibility to automatically provide more flexibility in their manipulation. To motivate this claim, we report in this chapter the results of an experimental study on the UML models publicly available on Github.[2] This study shows up to 93% of compatibility opportunities to load the models according to four subsequent versions of the UML standard. We also identify different intermediate flexibility levels according to the objective (support of the application of read-only model transformations, or in-place model transformations that modify the model) and the usage context (support of the compatibility to apply any model transformations, or for a particular model transformation whose footprint can be computed [143]).

To address these limitations, we propose to circumvent the overly restrictive conformance relation standing between models and metamodels with a dedicated typing relation. We explore this principle through a disciplined approach that leverages family polymorphism [92] to provide an advanced type system for manipulating models, in a polymorphic way, through different DSL interfaces. A DSL interface specifies a set of features, or services, available on the model it types. In our approach, these interfaces are captured in *model types* and supported by a typing relation between models and model types [239] and a subtyping relation between model types [119]. Model types are structural contracts over a language. They are used to define a set of constraints over admissible models, where a model is a graph of meta-class instances referred to as objects. Subtyping relations define the safe structural substitutions from one DSL to another. This opens up the possibility to define model manipulation tools (e.g., transformations, checkers, compilers) that can be reused for different languages, provided that they implement the required

[2] https://github.com/

interface. This is the approach followed by the Melange language workbench to support reusable DSL specifications [76], as we shall see in Chapter 6.

In this chapter:

- We list the properties of the conformance relation that are at the heart of the current MDE approaches and show how they hinder flexible modeling. We complete this study by analyzing the flexibility opportunities through an experimental study of the UML models publicly available on Github;
- We present the necessary concepts and relations for MDE that lift the current limitations by complementing the conformance relation with a typing relation based on an explicit structural language interface – model type, therefore providing increased compatibility and interoperability to manipulate a model through different DSL interfaces;
- We describe a model-oriented type system that supports these concepts and relations to provide a safe mechanism of polymorphism for models. This type system ensures *safe structural substitutability* of models conforming to different languages;
- We introduce Melange, a language workbench seamlessly integrated with the *Eclipse Modeling Framework* (EMF) ecosystem, with an implementation of the model-oriented type system. We detail how the implementation of Melange successfully emulates type groups polymorphism and structural typing to provide model polymorphism seamlessly on top of the legacy EMF ecosystem;
- We illustrate the validity and practicability of our approach through a controlled experiment on a family of finite-state machine languages, and an uncontrolled experiment on the UML models collected on Github.

5.2 On the Limits of the Conformance Relation

To state whether a model is a valid instance of a DSL, MDE relies on the *conformance relation* that stands between a model and its metamodel. The conformance relation plays a crucial role in MDE as it identifies *which* models are valid instances of a given DSL and *how* they should be manipulated. In this section, we first review the conformance relation definitions used in the literature, tools, and standards. Based on these definitions, we provide and detail theoretical limits of this conformance. We then conduct a systematic analysis of UML models gathered from the popular repository hosting service Github. Specifically, we show that the constraints enforced by the conformance relation can be relaxed to increase the level of flexibility in model manipulation without losing any guarantee in terms of safe model manipulation.

5.2.1 Limits of the Conformance Relation from a Theoretical Point of View

In MDE, the abstract syntax of DSLs is usually defined by a metamodel [231]. Based on this cornerstone artifact, concrete syntaxes, semantics, and various tools can be defined [240]. To determine whether a model is a valid statement of a DSL, MDE relies on the conformance relation that stands between a model and a metamodel. While no standard definition of the conformance relation exists, a recurring definition has emerged from the literature: a model conforms to a metamodel if every element of the model is an instance of one of the elements of the metamodel.

Different names have been given to this relation in the literature: "sem" (e.g., Bézivin and Gerbé [23]), "instantiation" (e.g., Atkinson and Kühne [7]) or "conformance" (e.g., Bézivin et al. [24]). All these relations are directly based on the abstract syntax of DSLs, expressed in the form of a metamodel. In the following, we use the term *conformance relation* to refer to this relation between models and metamodels. Table 5.1 presents several definitions of the conformance relation from the literature over the last decade.

Favre considers every representation of a language as a metamodel, and thus builds the conformance relation between a model and any of these representations (abstract or concrete syntax, documentation, tool, etc.) [94]. Favre's definition is thus less strict than the other ones presented in this section. However, while this definition is sufficient for the study and understanding of MDE, it is not precise enough for DSLs tooling or automated checking of the validity of a model wrt. a DSL.

Other authors define the conformance relation through the instantiation relation that stands between an object and the class from which it is built: "every element of an Mm-level model must be an instance-of exactly one element of an Mm+1-level model" [7]; "metamodels and models are connected by the instanceOf relation" [113]; "every object in the model has a corresponding non-abstract class in the metamodel" [86]. Bézivin and Gerbé do not directly refer to classes, but prefer the terms "definition" [23] or "meta-element" [24] (i.e., element of a metamodel). Such definitions authorize the definition of the abstract syntax of a DSL under a different form than a set of classes. With the exception of the definition given by Favre, definitions from the literature presented in Table 5.1 all agree on one point: the conformance relation is based on the instantiation relation between objects and classes.

The Meta-Object Facility (MOF) specification [204] from the Object Management Group (OMG) and the Eclipse Modeling Framework [240] (EMF) are *de facto* technological standards in the MDE community. Many tools and frameworks are based on these standards, such as ATL [150], Kermeta [147], Epsilon [164], or Xtext [93] to name just a few. The way these standards define the relation between models and metamodels is thus central in today's tooling support of MDE.

Bézivin and Gerbé [23]	"Let us consider model X containing entities a and b. There exists one (and only one) meta-model Y defining the "semantics" of X. The relationship between a model and its meta-model (or between a meta-model and its meta-meta-model) is called the *sem* relationship. The significance of the *sem* relationship is as follows. All entities of model X find their definition in meta-model Y."
Atkinson and Kühne [7]	"In an n-level modeling architecture, M0, M1...Mn-1, every element of an Mm-level model must be an instance-of exactly one element of an Mm+1-level model, for all m < n - 1, and any relationship other than the instance-of relationship between two elements X and Y implies that level(X)=level(Y)."
Favre [94]	Favre does not give a definition for conformance relation, but presents it as a shortcut for the sequence of two other relations: "elementOf" (which stands between a model of a language and this language) and "representationOf" (which stands between a metamodel and modeling language).
Bézivin et al. [24]	"A model M conforms to a metamodel MM if and only if each model element has its metaelement defined in MM."
Gasević et al. [113]	"metamodels and models are connected by the instanceOf relation meaning that a metamodel element (e.g., the *Class* metaclass from the UML metamodel) is instantiated at the model level (e.g., a UML class *Collie*)."
Rose et al. [226]	"A model conforms to a metamodel when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel. [...] For example, a conformance constraint might state that every object in the model has a corresponding non-abstract class in the metamodel."
Egea and Rusu [86]	"Namely, the objects of a "conformant" model are necessarily instances of the classes of the associated metamodel (possibly) related by instances of associations between the metamodel's classes."

Table 5.1: Definitions of the conformance relation in the literature

MOF permits the definition of the abstract syntax of DSLs in the form of an object-oriented metamodel. MOF, however, does not give any indication regarding the relation that stands between a model and a metamodel. Besides, UML is said to be an instance of MOF and "every model element of UML is an instance of exactly one model element in MOF" [207].

EMF does not give any definition either, but relies on two technologies to manipulate models: Java classes for instantiating elements of models and XML Schema for model serialization [240].[3] On the Java side, one class is generated for each concept of the abstract syntax and models are sets of object instances of these generated classes. On the XML side, an XML Schema describes the structure of a metamodel for enabling the serialization of models as XML documents. The XML Schema recommendation states that Conformance (i.e., validity) checking can be viewed as a multi-step process [267]: first, the root element of the document instance is checked to have the right contents; then, each sub-element is checked to conform to its description in a schema. Moreover, "to check an element for conformance, the processor first locates the declaration for the element in a schema" [267]. Thus, an element of an XML document without a corresponding declaration in a XML Schema does not conform to this schema, neither does the whole document. Metamodels being described through XML Schemas and models through XML documents, a model conforms to a given metamodel if all the elements of the model have a corresponding declaration in the XML Schema of the metamodel.

The three following major limitations of the conformance relation stand out from these definitions:

(1) The conformance relation is instantiation-based. The relation between a model and its metamodel is set up at the time the model is instantiated. In practice, a model is typically stored in an XML file, with an unchangeable and explicit URI that identifies the metamodel used to create it.

(2) The conformance relation is nominal. In the type system domain, nominal typing refers to a type system that relies on types' names to define explicitly the typing relations [99]. For instance, the Java language has a nominal type system: in "*class A extends B*", the keyword "*extends*" explicitly appoints B as the super type of A. By analogy, the conformance relation in MDE is nominal: a model explicitly refers to its "type" materialized in the URI that points to its metamodel.

(3) A model conforms to one and only one metamodel. This property is a consequence of the two previous ones. Because the conformance relation is instantiation-based, nominal, and only one metamodel is used to create a model, a model conforms to this metamodel only, throughout its lifetime.

[3] EMF can also instantiate models reflectively or using a dedicated serialization mechanism provided by users of the framework. However, we only consider the case of XML Schema which is the default behavior of EMF.

These three properties make explicit the metamodel that must be used to manipulate a model, thereby avoiding unsafe manipulations. The immediate drawback is that, by requiring one particular metamodel, it is not possible to use another one (even a close one, e.g., a subsequent version) to manipulate a model. For instance, when a language evolves, the URI of its metamodel is usually updated to materialize the new version. So, models, tools, and transformations defined over the previous version must be subsequently updated, even when the two versions are forward-compatible [171]. Some tools (e.g., ATL [150]) rely on dynamic typing mechanisms and are thus less fragile to evolution. In this case, however, it is not possible to determine *statically* whether a model can be manipulated by a given tool. The gains in terms of flexibility thus occur at the cost of safety.

One can astutely cope with this limitation by sharing the same URI among the versions of the language, even though they actually describe different languages. This, however, implies that language tools cannot determine *a priori* whether they will be able to process a model. This limitation also extends to languages that share commonalities modelers want to handle seamlessly. In other words, as we shall see in the next section, the strong insurances in terms of safety occur at the cost of flexibility.

5.2.2 Limits of the Conformance Relation from an Experimental Point of View: the Case of the UML Models on Github

UML is widely used for the object-oriented analysis and design of software systems [37]. Various tool providers provide their own implementation of the different versions of the UML specification. Within the Eclipse ecosystem, the Model Development Tools[4] sub-project (MDT) provides an EMF-based implementation of the UML2 specification, along with other closely related technologies such as the Object Constraint Language (OCL).[5] Each new revision of the UML specifications (every two or three years) leads to new major versions of the MDT-UML2 implementation. On UML metamodel changes, UML models need to be updated subsequently to take the novelties into account – even when changes in the metamodel do not directly impact them. To help modelers migrate their UML models, MDT-UML2 provides migration guides[6] that detail the changes for each new version. As the guides show, new versions usually *add*, *remove*, and *modify* elements, operations, and constraints of the metamodel, consequently breaking parts of the associated API. To cope with these changes, each guide usually describes a migration procedure that UML modelers should follow.

[4] http://www.eclipse.org/modeling/mdt/
[5] http://www.omg.org/spec/OCL/
[6] https://wiki.eclipse.org/MDT/UML2#Guides

In this section, we present a systematic analysis of UML models hosted on Github. We show that, although the conformance relation prevents it, most UML models can be loaded using different versions of the MDT-UML2 metamodel without having to be explicitly migrated. This requires to manually bypass the nominal typing constraint (i.e., the metamodel URI) imposed by the conformance relation.

Dataset Collection We collect an initial dataset of UML models by crawling Github repositories and gathering all the files having either the *uml* file extension, or the *xmi* file extension and containing the term *uml*. This query retrieves a total of 8737 files at the date of 2014-07-07. We first discard all the duplicates[7] and the models created using another modeling tool such as ArgoUML or Modelio, as they cannot be processed using MDT-UML2. At that point, we obtain 3647 models that specify one version of the MDT-UML2 metamodel as the URI of their metamodel. We then automatically process those models and prune from the dataset those that cannot be loaded because (i) their XMI serialization is ill-formed or (ii) they exhibit external dependencies towards other metamodels or custom UML profiles that cannot be systematically resolved. Table 5.2 depicts the results of this selection. A model is considered "loadable" if there exist at least one version of the MDT-UML2 metamodel that can be used to load it. In our experiment, we consider the UML2.2 to 2.5 specifications that correspond to the four major revisions of MDT-UML2 (2.x to 5.x). Besides the UML metamodel, MDT provides a validation framework that defines a set of constraints derived from the UML specifications and implemented as Java code. This validation framework ensures that a given model conforms to its metamodel and the associated constraints (i.e., its static semantics). We systematically run the validation framework on each loaded model and obtain 1651 valid models. In the remainder of this section, the set of valid models constitutes our base dataset for different analyses.

Extracted	Duplicates	Uniques	MDT-UML2	Valid
8737	2767	5970	3647	1651

Table 5.2: Initial dataset collection

We then identify for each valid model the precise metamodel version to which it conforms, i.e., the one used to create it. This information is directly extracted from their serialized form, as each of them contains an explicit URI

[7]Duplicates were mined and removed using the *fdupes* program that relies on full MD5 hashes and byte-to-byte comparison, i.e., syntactically and semantically equivalent models may not be detected as duplicates in this phase

referring to its precise metamodel. As depicted in Table 5.3, they cover all the major revisions of UML we are considering.

Version of UML	2.2	2.3	2.4	2.5	All
Conforming models	183	726	682	60	1651

Table 5.3: Distribution of the UML versions of valid models

Analysis We then analyze the resulting models with respect to the four major version of UML implemented in MDT-UML2, namely UML2.2 to 2.5. EMF normally relies on the URI stored in the XMI of a serialized model to determine the metamodel that should be used for loading and manipulating it. As an immediate consequence, it prevents loading the same model using different versions of a metamodel, as the URI is likely to change to reflect the current version of the metamodel (this is the case for UML). In this experiment, we purposely try to load each valid model with each different version of UML, even though EMF would usually prevent it. To do so, we change the URI stored in the models on-the-fly, just before invoking the parser. Then, for the models that are successfully loaded with a given metamodel, we run the corresponding validations. Table 5.4 sums up how many of the 1651 models can be loaded and validated with each different version. If a model makes use of a feature that does not exist in the metamodel used to load it, the parser may fail. For instance, a model that makes use of a feature introduced in UML2.4 cannot be loaded using the UML2.3 version of the metamodel. Similarly, constraints introduced, modified, or removed from a specific version can influence the result of the validation.

UML Version	2.2	2.3	2.4	2.5
Validated	1502	1497	1502	1460
% of all valid models	90.98	90.67	90.98	88.43

Table 5.4: Valid models per version

Number of compatible versions	1	2	3	4
Number of models	119	127	32	1373
% of all valid models	7.21	7.69	1.94	83.16

Table 5.5: Compatible versions per model

As shown in Table 5.5, only 7.21% of the UML models are strictly tied to a specific version of their metamodel, while 83.16% of them may be loaded and validated using any version. Overall, as shown in Table 5.4, each version of the UML metamodel can load around 90% of all the valid UML models, regardless of the original metamodel they conform to. These results can be explained by several factors. First, the EMF parser relies on an XML parser that only traverses the nodes present in the serialized model. This means that any model making use of features that are left unchanged across several metamodel versions can be loaded using any of these versions. Second, the parser creates in-memory model elements that are instances of the meta-classes of the metamodel it was configured to use, regardless of the meta-classes that were used to serialize the model. This means that, for example, an `Association` serialized using UML2.3 may be later parsed using UML2.4, provided that the features of `Association` used in the model did not change in the meantime. This implies that the subsequent validation phase will run just fine, as the manipulated types are the expected ones, thereby avoiding run-time errors. Naturally, validation may fail if the model does not meet the updated set of constraints.

This is similar to a common situation in the programming world where, for instance, the same `.cpp` file may be processed using various versions of a C++ compiler, implementing various versions of the C++ standard. A compiler implementing the C++03 standard may *try* to load a `.cpp` file written in C++11, but will ultimately raise an error if the file makes use of constructs specific to C++11. In this case, the gains in terms of flexibility occur at the price of safety, since it is no more possible to determine *a priori* if the program can be safely manipulated.

Admittedly, models found on Github may not always represent the state of the modeling practice in the industry. However, we find that the models we analyzed have an average size of 102 model elements and that their footprint [143] cover, on average, 40% of the UML metamodel. Moreover, we find that among the 1651 footprints extracted from the valid models, only 382 are unique. This is due to the fact that, for instance, UML models describing a class diagram share the same or similar footprints (the subset of UML necessary for expressing class diagrams).

5.2.3 Opportunities of Flexible Modeling Beyond the Conformance Relation

From this experiment, we envision that flexibility and safety in the manipulation of models are two frictional forces that must be constantly balanced. Safe manipulation of a model is highly dependent on the nature of the manipulations one would like to apply to it. Other authors have studied this notion of *usage context* in modeling [171]. While the conformance relation ensures that manipulating a model through its metamodel is always safe, regardless of the

context, it hinders flexible modeling by preventing the manipulation of models in other contexts where safety could still be ensured.

From our study of the UML models on Github, we see that flexibility can be drastically improved when the set of expected manipulations (in this case, parsing and validation) can be bounded. Naturally, the method employed in our experiment is not satisfactory, as it ultimately amounts to some kind of "Russian roulette": most of the time, models can be loaded safely, but when it is not the case the parser unpredictably fails – and the only way to find out is to actually try.

What is missing here is a relation that would state whether a model can be safely manipulated with respect to a set of expected manipulations or not. This relation would prevent unsafe manipulations and ensure that the right level of flexibility can be achieved for a given context. As an illustration, in the UML experiment, the only requirement is the read-only access to the subset of the metamodel that is actually used by the model (i.e., its footprint), hence the substantial reuse opportunities. Conversely, applying a transformation with side effects (e.g., an in-place transformation) to the model would require read-write access to its footprint, thereby strengthening the constraints on the model and restraining the opportunities of flexible manipulation. If the impact of all manipulations cannot be easily determined, the relation must ensure complete access to the entire model, which would further reduce the opportunities of flexibility, as the relation would have to take the whole metamodel into account. Finally, the guarantees that must be ensured through model manipulation may go up to behavioral substitutability, where every desirable property of the manipulating program should be kept [241].

As we are interested in model manipulation, we need to be able to express the minimal contract required to safely manipulate a model in a given context. In MDE, this contract is usually materialized in the metamodel defining the abstract syntax of a DSL. As we have shown, however, metamodels and the conformance relation are too restrictive to achieve the right level of flexibility. To alleviate these limitations, we propose to explicitly reify the contract imposed by a given context. This contract may be manually written or automatically inferred, for instance by extracting the metamodel footprint of a given model transformation [143]. This relaxes the constraints imposed by the conformance relation by enabling the safe manipulation of models fulfilling the appropriate contract, regardless of the metamodel used to create them.

We propose to reify such contracts as explicit structural interfaces expressed in the form of model types that abstract the possibly arbitrarily complex implementation techniques used to construct a DSL. Section 5.3 presents the structural interfaces we propose, along with the associated type system that supports flexible modeling through model polymorphism.

5.3 A Type System for Flexible Modeling

As shown in the previous section, the conformance relation hinders flexible modeling by preventing the manipulation of the same model in different modeling contexts or environments. In this section, we propose a model-oriented type system for flexible modeling that alleviate this limitation by enabling *model polymorphism*, i.e., the possibility to consider and manipulate a model under different forms. This type system relies on *explicit structural interfaces* captured in *model types* [239] that materialize the contract models must fulfill to be manipulated in a given context. Specifically, we show how to ensure *structural substitutability* between models that conform to different languages.

Section 5.3.1 introduces the necessary concepts and relations for separating implementations of languages from their structural interfaces and expressing such contracts. Section 5.3.2 presents the different subtyping relations we use to check the substitutability between model types and enable model polymorphism.

5.3.1 Reifying Model Types as Structural Language Interfaces for Model Polymorphism

The implementation techniques employed to define the different concerns of a DSL (e.g., syntax, semantics) are diverse and arbitrarily complex. For instance, different formalisms can be used to define a metamodel (e.g., an entity-relationship diagram, a class diagram, the MOF formalism) and to define the operational semantics on top of a metamodel (e.g., ATL transformations [150], Kermeta or K3 aspects [147], or simply Java code). To ease reasoning on language implementations, we propose to reify the concept of structural language interface and to explicitly separate language implementations from their interfaces. A structural language interface is similar to a metamodel as it exposes a set of concepts and their features and specifies *how* the models matching this interface must be manipulated, i.e., what is the contract they must fulfill. Contrary to metamodels, language interfaces are inherently abstract and cannot be used to instantiate models. We choose to use model types [239] as the formalism for expressing structural language interfaces. The benefit of model types compared to metamodels is that the same model can be manipulated through various model types thanks to subtyping relations [119]. Explicitly separating interfaces from implementations permits to use interfaces as first-class entities. Therefore, they can be used to explicitly state what is the contract a model must fulfill in order to be manipulated in a given context or environment.

Figure 5.1 presents an overview of the concepts and relations of the proposed type system and how they seamlessly integrate with the existing modeling concepts. These concepts and relations are detailed hereafter and are explained

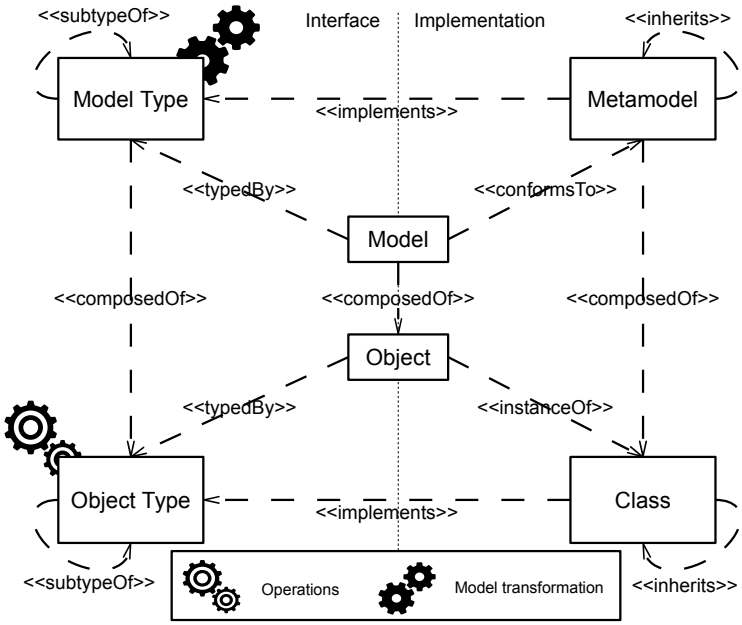
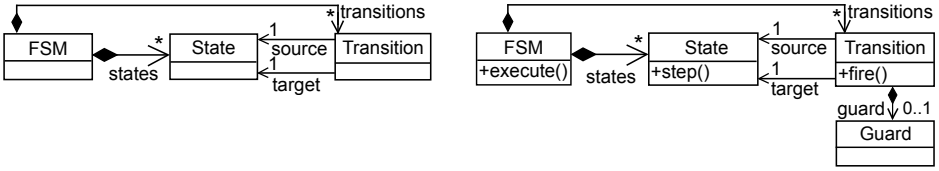


Figure 5.1: Separating implementations and interfaces of languages

using an illustrative example consisting of two variants of a finite-state machine (FSM) metamodel (Figure 5.2): a simple FSM (*Fsm*, Figure 5.2a) and an executable FSM with simple guards on transitions (*GuardFsm*, Figure 5.2b). These two variants define the concept of *FSM* composed of *State*s and *Transition*s. A *Transition* has a reference to its *source* and *target* states. Regarding *GuardFsm*, an FSM model can be executed (operations *execute*, *step*, and *fire*) and transitions can declare a *Guard*.



(a) The *Fsm* metamodel (b) The *GuardFsm* metamodel

Figure 5.2: The metamodels of two variants of a finite-state machine language

Object Types. Mainstream object-oriented programming languages provide developers with the concept of explicit interface (e.g., the `interface` keyword in Java). Such explicit interfaces permit the definition of types as contracts, usually consisting of a set of method signatures. This enables instances of a certain class to be manipulated through these interfaces, provided that the class implements the appropriate interfaces. An *object type* is an explicit structural interface: it exposes a subset of the features defined in the implementing

class and thus available on its instances. We use the term *object type* instead of *interface* to avoid any confusion with other uses of the term *interface* in this work. We graphically denote the concept of object type using the class representation supplemented with the symbol OT. Figure 5.3 shows an example of an object type of the *Transition* meta-class from the *GuardFsm* metamodel, that exposes only its *fire* method.

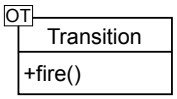
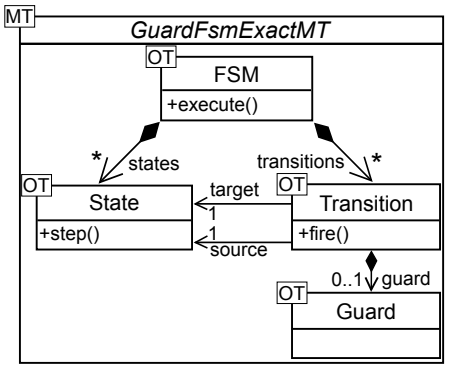


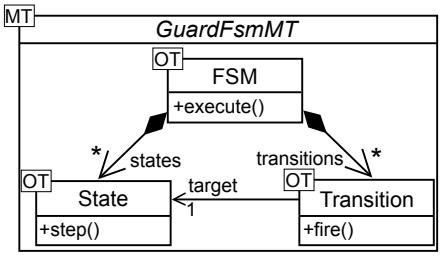
Figure 5.3: An object type of the *Transition* class

Model Types. A *model type* is an explicit structural interface that defines the contract a model must fulfill to be manipulated through the model type. Model types consists of a set of object types and their relations, and a model can be typed by multiple model types. A model type thus defines a *group of interrelated types*. To avoid unsafe manipulations, the consistency of type groups must not be ensured, i.e., types of different groups must not be mixed [92]. Among all the model types of a given model, its *exact model type* contains all the exact object types corresponding to the classes of its metamodel. The exact model type of a model is thus its most precise type. It can be directly extracted from the metamodel used to create the model.
Formally, the exact model type $Ex(MM)$ of models conforming to a metamodel MM is a model type such that $\forall T \in Ex(MM), \exists C \in MM$, and $\forall C \in MM, \exists T \in Ex(MM)$ such that $T = Ex(C)$.

Figure 5.4 illustrates the concept of model type using the FSM example. We graphically denote the concept of model type using the class representation supplemented with the symbol MT. The structure of a model type is graphically denoted by a class diagram of its object types. Figure 5.4a depicts the exact



(a) The exact model type of *GuardFsm*



(b) A model type of *GuardFsm*

Figure 5.4: Examples of model types of the *GuardFsm* metamodel

model type (named *GuardFsmExactMT*) of *GuardFsm*. *GuardFsmExactMT* exposes all the features of all the object types of *GuardFsm*.

Figure 5.4b shows an example of a model type (*GuardFsmMT*) of *GuardFsm*. *GuardFsmMT* does not expose all the features of all its object types. The *source* feature of the *Transition* object type and the *Guard* object type are omitted. Therefore, *GuardFsmMT* cannot be considered as the exact model type of *GuardFsm* but only as one of its model type.

Model Typing Relation. We call *model typing relation*, the typing relation that stands between a model and its model types. This typing relation brings flexibility against the standard conformance relation since it allows a model to have several model types. While each model element in a model is instance of only one specific class (defined in its metamodel), it can be typed by multiple object types (defined in different model types). However, because model types form a group of interrelated types, their consistency must be ensured and types of different groups must not be mixed [92]. Consequently, a model m is typed by a model type MT if all the model elements in the model are typed by an object type defined by MT.
Formally, the model typing relation (:) is a binary relation from the set of all models $\mathcal{M}$ to the set of all model types $\mathcal{MT}$, such that $m : MT$ iff $\forall o \in \mathcal{M}, \exists t \in MT$ such that o is typed by t, where $m \in \mathcal{M}$, and $MT \in \mathcal{MT}$.

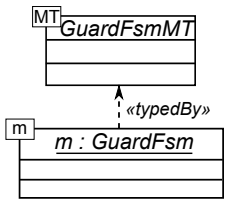


Figure 5.5: Model typing relation example

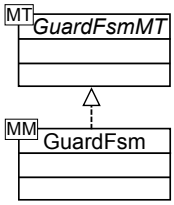


Figure 5.6: Implementation relation example

Figure 5.5 illustrates the model typing relation using the *GuardFsmMT* model type depicted in Figure 5.4a. We graphically denote a model using the standard object instance representation tagged with m. In Figure 5.5, m conforms to the metamodel *GuardFsm* and is thus typed by the model type *GuardFsmMT*.

Implementation Relation. An *implementation relation* stands between metamodels and model types. This relation specifies that a given metamodel MM provides the implementation of the features declared in a given model type MT. It means that any model conforming to MM can be manipulated through MT.
Formally, a metamodel MM implements a model type MT if for each object type in MT there is a corresponding meta-class in MM implementing it.

Figure 5.6 illustrates the implementation relation using the model type depicted in Figure 5.4b. We graphically denote the implementation relation that stands between a metamodel and a model type using the standard implementation representation. Figure 5.6 depicts a model type *GuardFsmMT* of *GuardFsm*. In this example, we graphically denote the concept of metamodel by tagging a class with the symbol MM. Using the proposed definition, *GuardFsm* implements the model type *GuardFsmMT* since all object types of *GuardFsmMT* have a corresponding implementation (a class) defined in *GuardFsm*.

The two previous definitions (model typing and implementation relations) are independent of different choices in the implementation of a model type checker [119].

5.3.2 Supporting Flexible Modeling with Model Polymorphism

In the previous section, we introduced the necessary concepts and relations for explicitly separating DSLs implementations from their structural interfaces. We capture these interfaces using model types, which express the contract a model must fulfill to be manipulated in a given context. In order to enable the manipulation of the same model through different interfaces, i.e., model polymorphism, a subtyping relation is needed to state in which cases substitutability is safe. This relation takes the form $\mathcal{MT} \times \mathcal{MT} \rightarrow Boolean$ where $\mathcal{MT}$ is the set of all model types. It states whether models typed by a model type can be substituted safely to models typed by another model type. Our approach do not assume a particular subtyping relation in general, but is instead parameterized by a chosen subtyping relation, which can vary according to particular needs [119]. In the remainder of this chapter, we focus on the *total isomorphic subtyping* relation introduced by Guy et al. This relation, denoted <:, checks subgraph isomorphism between the concepts (i.e., object types) of two model types, ensuring that all the concepts of the super-model type have a matching concept in the sub-model type [119]. The total isomorphic subtyping relation strengthen the model type matching relation proposed by Steel and Jézéquel [239] to prevent model manipulation from instantiating an element without its mandatory properties. Naturally, our approach remains applicable using other subtyping relations, such as the ones introduced by Guy et al. to support adaptation between two model types or partial subtyping. Furthermore, the concepts presented can be applied to support behavioral substitutability, e.g., by taking into account contracts expressed as pre- and post-conditions [241], or simulation relations based on event structures [174]. In this chapter, however, we focus on safe *structural* substitutability, and leave the deeper issue of behavioral substitutability to future work.

While the total isomorphic subtyping relation states whether structural substitutability between two model types is safe, regardless of the context, it hinders other scenarios of flexible modeling that arise when considering the context. For instance, as envisioned in Section 5.2.3, the additional constraint introduced by Guy et al. on elements instantiation can be relaxed when it is known that the manipulations have no side effects on their input models and do not instantiate new elements. To gather such information, one can extract the static footprint of the transformations she would like to apply [143]. In this case, the contract a model must fulfill in order to be manipulated through these transformations can be reified as a model type corresponding to the footprint. The substitutability between two model types in the context of a transformation t can then be formulated as $MT' <: fp(t, MT)$ where $MT, MT' \in \mathcal{MT}$ and $fp(t, MT)$ is the footprint of the transformation t on MT.

Statically inferring the footprint of a transformation may not always be possible when the transformation is black-box or because of the cost of static analysis; and is imprecise at best. To improve the preciseness of footprints, one can rely on dynamic footprinting [143]. In this case, the transformation is invoked on a model of interest while recording a trace of its execution. Doing so, the footprint is much more precise but the benefits of static checking are lost, and substitutability between model types cannot be checked statically.

5.4 Implementation of Model Polymorphism in Melange

Melange[8] is an open-source language workbench bundled as a set of Eclipse plug-ins. Melange provides support for executable and aspect-oriented metamodeling, along with operators for DSL assembly and customization (cf. Chapter 6). We refer the reader to Chapter 7 for an in-depth presentation of Melange and K3. In this section instead, we focus on its support for flexible modeling through the type system introduced in Section 5.3 and a dedicated mechanism named the *MelangeResource*. We briefly present the syntax of Melange for defining languages and model types (Section 5.4.1). We then put the emphasis on its support for model polymorphism (Section 5.4.2) and seamless integration with the EMF ecosystem (Section 5.4.3).

5.4.1 Language and Model Type Definition in Melange

Melange relies on various meta-languages for expressing the different concerns that compose a DSL, in particular Ecore [240] (provided by the EMF framework) for defining their abstract syntax in the form of metamodels and K3 for defining

[8] http://melange-lang.org

their operational semantics as a set of aspects [147]. The choice of Ecore is motivated by the success of EMF both in the industry and academic areas.

Melange comes with a textual editor that enables the definition of DSLs using a dedicated syntax. The minimal definition of the *GuardFsm* language introduced in Figure 5.2b is given in Listing 5.1. The **syntax** keyword specifies the Ecore file that defines its abstract syntax. The **with** keyword is used to weave the aspects defining its operational semantics. Finally, the **exactType** keyword automatically extracts from its implementation its exact model type *GuardFsmMT* which exposes both the features defined in its metamodel and the features woven by aspects. Every language definition in Melange must specify its exact type using the **exactType** keyword. Listing 5.2 illustrates the explicit definition of the exact model type of *Fsm* depicted in Figure 5.2a. The **modeltype** keyword explicitly defines a model type to enable fine-grained control over the exact required set of features for a specific purpose (e.g., opening a model in a specific environment or applying a given transformation). Nonetheless, *FsmMT* and *GuardFsmMT* share the same nature: they are structural language interfaces expressed in the form of a model type.

```
1 language GuardFsm {
2   syntax "GuardFsm.ecore"
3   with ExecutableFsm
4   with ExecutableState
5   with ExecutableTransition
6   exactType GuardFsmMT
7 }
```

Listing 5.1: The *GuardFsm* language

```
8  modeltype FsmMT {
9    syntax "FsmMT.ecore"
10 }
11 language CustomFsm
12   implements FsmMT {
13   [...]
14 }
```

Listing 5.2: The *FsmMT* model type

From a Melange specification, such as the one presented in Listings 5.1 and 5.2, the type system of Melange automatically infers the subtyping relations among the declared model types and the implementation relations between languages and model types, as specified in Section 5.3. To this end, the model-oriented type system of Melange relies on the total isomorphic subtyping relation introduced by Guy et al. [119]. Naturally, every language directly implements its exact model type (e.g., *GuardFsm* implements the *GuardFsmMT* model type). Subtyping relations among model types are also inferred in this phase. For instance, in this case the type checker infers that *GuardFsmMT* is a subtype of *FsmMT* since all the features of the latter are also present in the former. It is worth noting that our implementation of the type system relies on structural typing by default: the type system analyses the structure of metamodels and model types to determine the typing relations, without requiring the user to explicitly specify the typing relations (as with nominal typing). However, users can require that a language implements a specific set of model types using the **implements** keyword (e.g., Line 12 of Listing 5.2). In

this case, the type checker reports an error to the user as long as the language does not implement one of its interfaces. This form of nominal typing enables a simple kind of design-by-contract in the case where the language designer knows, at design time, in which environments models conforming to it should be manipulated. Finally, Melange enables to fix simple structural dissimilarities by allowing users to rename concepts in a metamodel or model type using a `renaming` clause. As we are interested in flexibility between structurally similar languages, we do not detail this mechanism further.

5.4.2 Implementation of the Support for Model Polymorphism

With the conformance relation, models can only be manipulated through their original metamodel. The core idea of model polymorphism is to supersede the conformance relation with typing relations that enable the manipulation of models through different interfaces materialized by model types. We detail in this section the mechanism we use to enable model polymorphism on top of the EMF framework. EMF is a Java framework, which does not provide any support for type groups polymorphism and structural typing. We detail hereafter how Melange successfully emulates such concepts atop the EMF framework.

For each meta-class in a Ecore metamodel, EMF generates a corresponding Java interface that materializes its features (e.g., an attribute is implemented as a pair of getters/setters). It also generates a Java class implementing the interface where actual values of the persistent features are stored in memory. Therefore, each model element in a model is an instance of the Java class generated from its meta-class. The whole model is a graph of objects that can be manipulated using the generated Java types. Along the generated Java types, EMF also generates a dedicated factory [110], responsible for the creation of new model elements. Any transformation that encompasses the creation of new model elements must use the generated factory.

Melange uses a similar mechanism for materializing model types at the Java level. For each model type, Melange leverages the EMF generator to generate the corresponding Java interfaces. These interfaces are used to manipulate a model in the same way one would manipulate a model through its metamodel. From a user's point of view, there is no difference between manipulating a model through a metamodel or through a model type. The Melange's compiler also generates an abstract factory declaring methods for creating new elements of the model type. Because model types are inherently abstract, no classes implementing them are generated. Instead, concrete implementations are provided by the languages that implement a model type, thereby enabling the manipulation of models written in a given language through the model types it implements, i.e., model polymorphism.

The set of all possible model types implemented by a given language are, however, not known at the creation of a language; typing relations are only inferred *a posteriori* by the structural type checker of Melange. So, there is no direct relation between the Java classes that form a language and the Java interfaces that form a model type. To solve this problem, we employ the *adapter* design pattern [110] to create the implementation relations between classes and interfaces *a posteriori*. The general idea is to generate a set of adapters for each pair ⟨*language, implemented model type*⟩ (one per object type in the model type). Each adapter implements an object type and delegates the implementation of its features to the corresponding class in the implementing language. Along the adapters for model elements, Melange generates a concrete factory that provides the implementation of the abstract factory of the model type for a given language. For each creation method in the abstract factory, the concrete factory delegates to the factory of the underlying language and encapsulates the result in an appropriate adapter. Melange uses a generative approach to generate the code of adapters by the time the implementation relation between languages and model types are inferred. The generation of these adapters is safe since the type checker ensures that each object type and feature in a model type has a corresponding implementation in the language.

The generated adapters ensure several properties. First, the graph of all adapters for manipulating a given model through a given model type is built lazily. Only the adapter of the root model element is initially created. Then, adapters of the other elements are built on demand when navigating references from one object to the other. For example, in Figure 5.4b, *State* adapters are only generated when navigating the *states* reference from *FSM* to *State*. Second, because a model type constitutes a family of types, generated adapters ensure that the semantics of type groups is respected, i.e., elements of different type groups cannot be mixed together. This constraint is inherited from family polymorphism [92] to ensure safe model polymorphism. Adapters also support dynamic dispatching by default. For instance, when a model type exposes an operation in one of its object type, calling this operation on a model will dynamically dispatch to the implementation provided by the actual language of the model. To limit memory overhead, the framework ensures that, for each model element, there is at most one adapter in memory – they are cached and retrieved whenever needed. Finally, adapters support model polymorphism through both direct manipulation and reflective manipulation of models using the reflective API provided by EMF [240]. Supporting polymorphism through the reflective API is especially important as many tools of the EMF ecosystems (e.g., Sirius [85], ATL [150]) rely heavily on reflection. Overall, our adapter approach allows to seamlessly manipulate models through model types, without disrupting language engineers habits in EMF. This is in contrast with other approaches providing similar benefits, such as *concepts* [70], which require to explicitly parameterize metamodels with explicit template parameters. We

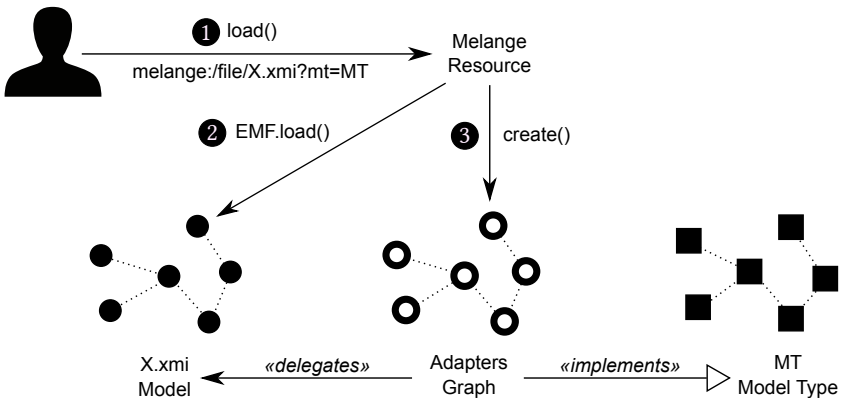


Figure 5.7: Leveraging the adapter pattern to polymorphically load a model as a specific model type

refer the reader to Listing A.5 to get a feel of the complete source code of an adapter.

5.4.3 Seamless Integration with EMF

Melange aims at providing model polymorphism for EMF-based languages and tools in a seamless and non-intrusive way: the models, their metamodels, and the transformations manipulating them must remain unchanged. To do so, Melange provides a dedicated mechanism, named the *MelangeResource*, that allows to specify in which context (i.e., through which model type) a model must be loaded. The same model can thus be loaded in different environments if its metamodel implements the appropriate model types.

EMF relies on the concepts of *resources* and *resource sets* to load serialized models in memory and save them back as persistent document [240]. A resource represents a persistent model and, when loaded, provides access to the model elements it contains. Resources are created using dedicated *resource factories* responsible for loading a particular kind of models stored in a particular format. Each resource is identified by a unique Unified Resource Identifier (URI) [19] that locates it on the file system (e.g., `file:/path/Model.guardfsm`). When a user requests the loading of a model, the file extension and protocol of its URI are analyzed to determine the appropriate resource factory that must be used to instantiate a new resource representing the model in memory. Then, the resource factory identifies the metamodel of the loaded model (from the nominal reference stored in the serialized model) and uses the metamodel's factory to create the appropriate AST nodes and obtain the graph of objects representing the model in memory.

Melange provides a specialized resource mechanism named the *MelangeResource* to seamlessly support model polymorphism on top of the EMF framework. Melange contributes a new *protocol parser* to EMF that automati-

cally delegates model loading to the *MelangeResource* when the protocol of its URI is `melange` (e.g., `melange:/file/path/Model.guardfsm`). When the user specifies the `melange` protocol the *MelangeResource* is used. In this case, the user can adjoin an additional query string parameter named `mt` to the URI. The `mt` parameter specifies the model type that is *expected*, regardless of the actual metamodel of the model. The *MelangeResource* ensures that the model can safely be loaded through this model type based on the typing relation inferred earlier. Internally, the *MelangeResource* instantiates the appropriate adapters that enable the manipulation of the model as typed by the expected model type. For example, the URI `melange:/file/Model.guardfsm?mt=FsmMT` specifies that the model stored in the `Model.guardfsm` file should be loaded as a (i.e., typed by) *FsmMT* model type. If no implementation relation between the metamodel of the loaded model and the requested model type exists, an error is reported to the user.

The benefit of using a Melange URI is that neither the model nor its metamodel or the transformation code has to be changed. Only the *inputs* of the transformations are modified: the URI of the models to be manipulated. Listing 5.3 depicts the typical code used to load a model using the EMF framework, with or without the *MelangeResource*. The only visible difference is the URI used to load the model. The model polymorphism mechanism and its runtime support (i.e., the adapters and the specialized resource system) are completely transparent for the user. In this case, the root of the model (Line 5) is returned as typed by the *FSM* object type defined in the *FsmMT* model type, even though the concrete type of the root of the model is the meta-class *FSM* defined in *GuardFsm*.

```
1 ResourceSet rs = new ResourceSetImpl();
2 //String oldUri = "file:/Model.guardfsm";
3 String uri = "melange:/file/Model.guardfsm?mt=FsmMT";
4 // Requests the model serialized at the given URI
5 Resource res = rs.getResource(URI.createURI(uri), true);
6 // Retrieve the first element of the model (ie. its root)
7 // getContents() is a generic function of EMF,
8 // the cast thus cannot be avoided
9 FSM root = (FSM) res.getContents().get(0);
```

Listing 5.3: Loading a model using a Melange URI

5.5 Experiments

In this section, we evaluate our approach for safe model polymorphism on the two axes of flexible modeling we consider: compatibility between subsequent versions of the same language and interoperability between structurally similar languages. We show that the type system described in Section 5.3

and implemented in Section 6.4 within Melange provides safe and seamless model polymorphism for EMF-based languages and tools. Section 5.5.1 shows how the high level of flexibility envisioned for UML models in Section 5.2.2 can be achieved with the *MelangeResource*. Section 5.5.2 shows how model polymorphism supports flexible model loading and manipulation within a family of related DSLs describing variants of finite-state machines.

5.5.1 Flexible Loading of UML Models

In Section 5.2.2, we showed that the conformance relation hinders many loading opportunities for UML models: 93% of the models we analyzed could be loaded and validated using at least two versions of the UML metamodel when forcing EMF to bypass the conformance relation. The technique used was however not satisfactory as the parser were unpredictably failing when a model could not be loaded with a given version of UML.

In this experiment, we aim to achieve the same scores in terms of flexibility while avoiding the "Russian roulette" downside. To this end, we leverage the information we gathered from the kind of manipulations applied to the loaded models. Only the meta-classes corresponding to the model elements they contain are used by the parser. If changes between two versions of the UML metamodel do not affect the subset of the metamodel used in a model, it can be loaded using both of them. Similarly, when invoking the validator, only the elements created by the parser are visited.

In consequence, while the footprint statically computed for the loading mechanism and the validator potentially corresponds to the entire metamodel, we can restrict this footprint to the meta-classes corresponding to the elements contained in the model (i.e., dynamic footprinting [143]). In other words, the *contract* an UML model must fulfill in order to be safely loaded and validated by a given UML version can be captured in a model type corresponding to the subset of the meta-classes actually required to type the elements it contains.

We use Kompren [28] to generate a pruner [234] for the different versions of the UML metamodel. Then, we visit each model to extract the meta-elements it uses and prune the resulting metamodel with respect to each version of the UML metamodel we consider. As a result, we obtain the effective footprint of each model on each version of UML. The resulting footprint corresponds to the dynamic footprint of the loading and validation phase of UML models. Then, when trying to validate a model with respect to a particular version of UML, we use the subtyping relation introduced in Section 5.3.2 to find whether the target UML metamodel is a subtype of the computed footprint. When it is the case, the model can be safely loaded and validated; otherwise, the program prevents subsequent crashes and reports an error to the user. The experiment results show that using this technique, we obtain the exact same scores as those envisioned in Section 5.2.2.

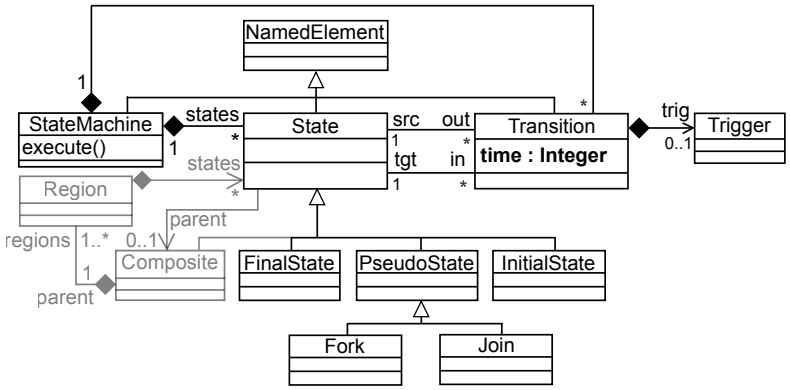


Figure 5.8: Excerpt of the metamodels of the FSM language variants

5.5.2 Model Polymorphism for a Family of DSLs

Finite-state machine (FSM) languages are a typical example of DSLs used in a wide range of contexts (e.g., systems and software engineering [122], language processing [223], user interfaces [26]). This leads to a rich diversity of implementations that exposes syntactic and semantic variation points [63]. Syntactic variants comprise flat or composite states, presence of temporal constraints or complex guards, etc. Semantic variants comprise different models of execution, e.g., with or without run-to-completion [114]. DSL designers usually design dedicated environments for each of these variants. Because of the conformance relation, there is a strong coupling between models and the modeling environments used to create them. So, DSL users cannot benefit from tools and transformations (e.g., editors, simulators, or code generators) integrated in different environments.

To solve this problem, a language designer can leverage model polymorphism as promoted in this work: a language designer can design a generic tool or transformation based on a *model type*. The model type makes explicit the required set of features that a model must provide (i.e., the contract it must fulfill) in order to be manipulated by this specific tool. So, any model typed by this model type can then benefit of this tool, regardless of the actual language that was originally used to create it.

In this case study, we show the benefits of model polymorphism on a family of FSM languages. On the syntactic side, the FSM languages differ in their support for composite states, time constraints, both, or none. On the semantic side, they differ on whether they implement a run-to-completion or simultaneous events processing semantics. Altogether, by combining syntactic and semantic variations, our final family comprises eight variants. These languages are representative of the actual variation points that are exposed by popular languages enabling the expression of FSMs such as UML, Rhapsody, and classical statecharts [63].

Figure 5.8 presents the variations in the abstract syntax of the languages of the FSM family. The gray part (the `Region` and `Composite` classes) is specific to the hierarchical FSM metamodels. The attribute `time` of `Transition`, formatted in bold font, is specific to the timed FSM variants. For the sake of conciseness, the other attributes are not represented.

We define each language in Melange. We create four Ecore metamodels representing the four syntactic variants of the family. The two semantic variants consist of two sets of aspects we wrote in K3 that define the operational semantics associated to run-to-completion and simultaneous events processing (see Section 5.4.1). Listing 5.4 illustrates the definition of two of the eight variants. The complete Melange file comprising the eight FSM variants is given in Listing A.1. Melange takes care of assembling a particular syntax with a particular semantics to produce the expected language. Note that in this case, most of the syntax and semantics is duplicated from one language to the other. One way to cope with this duplication is to leverage the composition operators of Melange presented in Chapter 6. Each language declares its `exactType` (Lines 7 and 18), i.e., the precise model type that represents the contract implemented by the models conforming to it.

```
1  // Flat state machine complying to the
2  // run-to-completion policy, e.g. UML/Rhapsody
3  language FlatFsmRtc {
4      syntax    "metamodels/FlatFsm.ecore"
5      with      rtc.ExecutableStateMachine
6      with      rtc.ExecutableState
7      exactType FlatFsmRtcMT
8  }
9
10 // Hierarchical state machine complying
11 // to the simultaneous events processing
12 // policy, e.g. Classical statecharts
13 language HierarchicalFsmSimultaneous {
14     syntax    "metamodels/HierarchicalFsm.ecore"
15     with      simultaneous.ExecutableStateMachine
16     with      simultaneous.ExecutableState
17     with      simultaneous.ExecutableTransition
18     exactType HierarchicalFsmSimultaneousMT
19 }
```

Listing 5.4: Two of the eight variants of finite-state machine languages

The type checker of Melange automatically infers the subtyping hierarchy among the exact model types, and the implementation relations between metamodels and model types. Based on the resulting hierarchy, the code generation phase generates the adaptation code supporting model polymorphism between the compatible variants (see Section 5.4.2).

Then, we implement two typical transformations on FSMs: *execute* checks whether a given sequence of events is recognized by a particular FSM model; *flatten* produces an equivalent FSM model without composite states. The former is defined over the most general model type `FlatFsmRtcMT` and can thus be polymorphically invoked on models conforming to any of the eight variants, taking into account the semantic variations thanks to the dynamic dispatch. The latter is defined over the model type of hierarchical FSMs and can thus be polymorphically invoked on models conforming to the four hierarchical variants.

```
1  // Delegate the execution of the state machine "fsm"
2  // to the "execute" method of its operational semantics.
3  // StateMachine is the root object type of FlatFsmRtcMT
4  public void execute(StateMachine fsm, String input) {
5    // Dynamically dispatched on the actual
6    // language's implementation of execute()
7    root.execute(input);
8  }
9
10 List<String> models = new ArrayList<String>();
11 models.add("melange:/file/m1.flat?mt=FlatFsmRtcMT");
12 models.add("melange:/file/m2.timed?mt=FlatFsmRtcMT");
13 models.add("melange:/file/m3.hiearchical?mt=FlatFsmRtcMT");
14 models.add("melange:/file/m4.timedhierarchical?mt=FlatFsmRtcMT");
15 ResourceSet rs = new ResourceSetImpl();
16
17 // Load the model pointed by the given URI,
18 // retrieve its root StateMachine, and execute it
19 for (String uri : models) {
20   Resource res = rs.getResource(uri, true);
21   StateMachine root = (StateMachine) res.getContents().get(0);
22   execute(res, "{x;y;z;o;p;q}");
23 }
```

Listing 5.5: Polymorphically invoking the *execute* transformation

Listing 5.5 shows the pseudo-code required for specifying the *execute* transformation in Java and calling it. The `fsm` parameter of the transformation is typed by the root object type `StateMachine` of `FlatFsmRtcMT`, and can thus be polymorphically invoked on models conforming to any of the eight variants. In this case, the execution semantics is directly woven using the aspects depicted in Listing 5.4: the *execute* methods simply delegates to the appropriate *execute* method of `StateMachine` through dynamic dispatch. However, manipulations in *execute* may be arbitrarily complex and use all the features depicted in Figure 5.8. The *MelangeResource* is automatically invoked by the use of a `melange:` URI and transparently instantiates the

appropriate adapter to enable the manipulation of the different models through the common interface `FlatFsmRtcMT`.

```
1  module FlattenFsm;
2  create OUT : FlatFsm from IN : CompositeFsmMT;
3
4  rule SM2SM {
5    from sm1 : CompositeFsmMT!StateMachine
6    to   sm2 : FlatFsm!StateMachine
7  }
8  -- Initial states of composite states become regular states
9  rule Initial2State {
10   from is1 : CompositeFsmMT!InitialState (
11     not is1.parentState.oclIsUndefined() )
12   to is2 : FlatFsm!State (
13     stateMachine <- is1.stateMachine,
14     name <- is1.name )
15 }
16 -- Resolves a transition originating from a composite state
17 rule T2TB {
18   from t1  : CompositeFsmMT!Transition,
19        src : CompositeFsmMT!CompositeState,
20        trg : CompositeFsmMT!State,
21        c   : CompositeFsmMT!State (
22          t1.source = src and
23          t1.target = trg and
24          c.parentState = src and
25          not trg.oclIsTypeOf(CompositeFsmMT!CompositeState) )
26   to t2 : FlatFsm!Transition (
27     name <- t1.name,
28     stateMachine <- t1.stateMachine,
29     source <- c,
30     target <- trg )
31 }
```

Listing 5.6: Excerpt of a generic ATL *flatten* transformation

Listing 5.6 depicts an excerpt of the *flatten* transformation written in ATL [150]. Most of the transformation rules are omitted for the sake of conciseness. In this case, the transformation requires as input a model typed by the `CompositeFsmMT` model type and produces a corresponding flattened state machine conforming to the `FlatFsm` metamodel. The *flatten* transformation thus accepts models conforming to any of the four hierarchical variants. One can write an ATL transformation on a model type in the exact same way than on a metamodel. When invoking the transformation the use of a Melange URI automatically invokes the *MelangeResource* so that the ATL engine transparently manipulates the models through the appropriate adapters. Making

an ATL transformation generic only requires to change the type of its input models to the appropriate model type.

5.5.3 Discussion

Through the two presented experiments, we illustrate the benefits of safe model polymorphism on the two axes of flexible modeling we consider: compatibility between subsequent versions of a same language; interoperability between structurally similar languages. Our framework and its implementation in Melange through the *MelangeResource* allows to state in which cases a model can be safely manipulated, without requiring any work from the language designers or users. As mentioned in Section 5.3.2, our framework is parameterized by a particular subtyping relation between model types. In our experiments, we use the total isomorphic subtyping relation introduced by Guy et al. [119]. As a result, our experiments only focus on structural substitutability and do not consider the harder problem of behavioral substitutability. For instance, while the *MelangeResource* ensures that the *flatten* transformation presented in Section 5.5.2 can safely be applied on a model conforming to different languages, it cannot state whether its behavioral properties will be preserved (e.g., are the resulting state machine models flat and semantically equivalent to the inputs models?). Other experiments involving augmented model types on which behavioral contracts are expressed in the form of invariants, pre-, and post-conditions to assess property preservation through the use of the contract-aware subtyping relation introduced by Sun et al. [241] are left for future work. Finally, we envision that the overhead in terms of time and memory consumption implied by our generative approach relying on adapters is highly dependent on the size of the considered models and metamodels.

5.6 Conclusion

In this chapter, we proposed to overcome the limitations of the conformance relation that hinder the flexibility DSL users would expect between structurally similar DSLs. We described a model-oriented type system that provides a safe mechanism of polymorphism for models. This type system is implemented into Melange, which is itself integrated with the EMF ecosystem. Through an experiment on UML models gathered from Github and a case study on a family of syntactically and semantically variant FSM languages, we showed the benefits of safe model polymorphism on the two axes of flexible modeling we consider: compatibility between subsequent versions of the same language and interoperability between structurally similar languages.

Chapter 6

Modular and Reusable Development of DSLs

In this chapter, I present our third contribution: a meta-language for modular and reusable development of DSLs. I introduce in Section 6.1 the context and motivation of our contribution. In Section 6.2, I give a general overview of our proposal, further detailed with an algebra of operators for DSL assembly and customization in Section 6.3 and its implementation in the Melange language workbench in Section 6.4. In Section 6.5, I evaluate the benefits of Melange by assembling and customizing several DSL implementations gathered from publicly available repositories to create a new DSL for Internet of Things systems modeling. Finally, I conclude in Section 6.6 on the observed benefits of Melange.[1]

6.1 Introduction

Despite the wide range of domains in which DSLs are used, many of them share commonalities such as a common paradigm for expressing workflows, components, or expressions [218]. It is long recognized that language designers benefit from reusing existing language definition for the creation of new languages [176, 88]. Similarly, many DSLs exist in different variants, such as the family of languages for statecharts modeling [63]. Therefore, it is likely that engineering efforts spent on the development of existing languages could be leveraged in the development of new ones. Following the time-honored practice of reuse in software engineering, one would like to reuse and combine existing language artifacts (e.g., a particular action language) in the development of new DSLs. Recent work in the community of Software Language Engineering focused on language workbenches that support the modular design of DSLs, and the possible reuse of such *modules*, usually using a scattered clause `import` linking separate artifacts (see e.g., [155, 249]). Besides, particular composition

[1] The material presented in this chapter led to the following publication: [76].

operators have been proposed for unifying or extending existing languages (cf. Chapter 3). However, while most of the approaches propose either a diffuse way to reuse language modules, or to reuse as is complete languages, there is still little support for easily assembling language modules with customization facilities (e.g., restriction of expressiveness or semantics specialization) in order to finely tune the resulting DSL according to the language designer's requirements or the specificities of a new domain of application. In addition, being able to reuse the services (e.g., editors, static analyses) accompanying an existing DSL would be highly beneficial for language designers. This issue is however seldom addressed by current approaches.

In this chapter, we present a tool-supported meta-language supporting the assembly and customization of legacy DSLs to produce new ones. This meta-language consists of a set of integrated operators dedicated to language manipulation. The operators enable to (i) merge different languages (ii) inherit from one language to the other (iii) restrict the scope of an existing language and (iv) finely customize the syntax and semantics of assembled languages. These operators can be used conjointly to solve advanced language engineering scenarios involving the composition of several languages. The meta-language is implemented in the Melange language workbench. Melange provides specific constructs to assemble various abstract syntax and operational semantics artifacts into a DSL. DSLs can then be used as first-class entities to be reused, extended, restricted, or adapted into other DSLs. Melange relies on the structural interfaces introduced in Chapter 5 to statically ensure the structural correctness of the produced DSLs, and subtyping relations between DSLs to reason about their substitutability and the reuse of services from one language to the other. Newly produced DSLs are correct by construction, ready for production (i.e., the result can be deployed and used as is), and reusable in a new assembly. We illustrate the benefits of the proposed meta-language by designing a new executable modeling language for the Internet Of Things domain. Specifically, we show how Melange eases the definition of new DSLs by maximizing the reuse of legacy artifacts without introducing issues in terms of performance, technical ecosystem compatibility, or generated code volume.

6.2 Approach Overview

As introduced in Chapter 2, DSLs are typically defined through three main concerns: abstract syntax, concrete syntax(es), and semantics. Various approaches may be employed to specify each of them, usually using dedicated meta-languages [259]. The abstract syntax specifies the domain concepts and their relations, and is defined by a metamodel or a grammar. The semantics of a DSL can be defined using various approaches including axiomatic semantics, denotational semantics, operational semantics, and their variants [194]. Concrete syntaxes are usually specified as a mapping from the abstract syntax

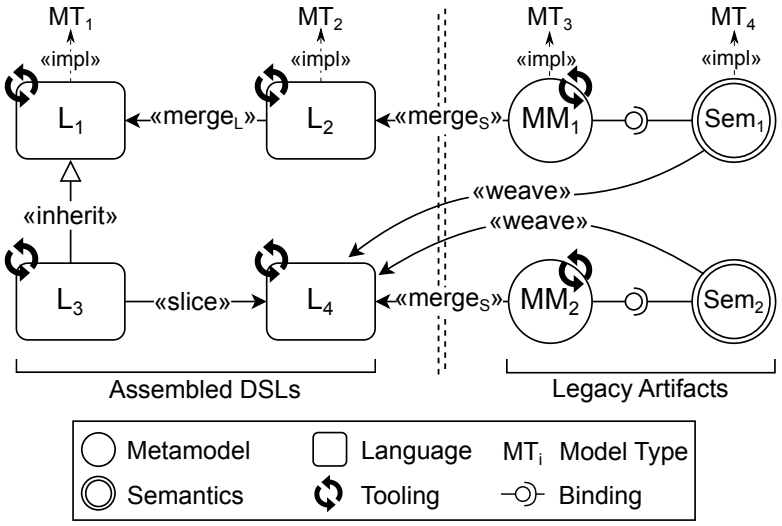


Figure 6.1: Assembling and customizing DSLs from legacy artifacts

to textual or graphical representations, e.g., through the definition of a parser or a projectional editor [265].

In this chapter, we focus on DSLs whose abstract syntaxes are defined with metamodels and whose semantics are defined in an operational way through the definition of computation steps designed following the interpreter pattern [110]. Computation steps may be defined in different ways, e.g., using aspect-oriented modeling [146] or endogenous transformations [59]. In this chapter, however, we only focus on the weaving of computation steps in an object-oriented fashion with the interpreter pattern. In such a case, specifying the operational semantics of a DSL involves the use of an action language to define methods that are statically introduced directly in the concepts of its abstract syntax, i.e., the meta-classes of its metamodel [147]. It is worth noting that the proposed approach can easily be adapted to other kinds of operational semantics specification mechanisms, such as endogenous transformations in a functional way. We do not address in this chapter the problem of concrete syntaxes composition and customization. The remainder of this section presents the high-level operators we propose for assembling and customizing DSL, as depicted in Figure 6.1.

The right part of Figure 6.1 depicts the legacy language artifacts to be reused and assembled in the creation of new DSLs. Imported artifacts include abstract syntax and semantic specifications, possibly with their corresponding tools and services (e.g., checkers, transformations). These services manipulate the models conforming to a particular metamodel to perform specific tasks. Similarly, semantic specifications directly access and manipulate model elements for execution or compilation purposes. Hence, abstract syntax and semantics artifacts are related one another through *binding* relations: semantics artifacts

require a particular shape of abstract syntax, which is *provided* by a given metamodel.

The left part of Figure 6.1 depicts the new languages built from legacy artifacts. *Assembly operators* ($merge_S$, *weave*) realize the transition from legacy artifacts to new DSLs. They import and connect disparate language artifacts, e.g., by merging different abstract syntaxes or by binding an existing semantics to a new syntax. Naturally, the same artifact can be reused in different assemblies. The output of assembly operators is encapsulated in a *language definition*. Once new assemblies are created, *customization operators* (*slice*, $merge_L$, *inherits*) offer the possibility to refine the newly built DSLs so as to meet additional requirements or to fit a specialized context. These customization operators closely match the taxonomy of language operators introduced by Erdweg et al. [90]: the *slice* operator supports *language restriction*, the $merge_L$ operator supports *language unification*, and the *inherits* operator supports *language extension*. We describe in detail all the operators for language assembly and customization in Section 6.3.

Assembling and customizing DSLs is a challenging task that requires checking the composability of heterogeneous artifacts and the validity of the result. For example, from Figure 6.1, it is clear that Sem_1 can be woven on L_4 only if it can be bound to its syntax, i.e., if the elements originally provided by MM_1 to Sem_1 are also available in L_4. Similarly, the intuitive meaning of inheritance, as found in most object-oriented languages, implies the compatibility between the super- and sub-elements (super- and sub-languages in this case). It follows that the compatibility between L_1 and L_3 in Figure 6.1 must be ensured to guarantee that L_1's tooling can be reused for its sub-language L_3. What is missing here to guarantee these properties is an abstraction layer that would support reasoning about the compatibility between different languages artifacts. In our approach, we rely on the capabilities provided by model types, as presented in Chapter 5, that provide an abstraction layer to reason about the structural compatibility and substitutability of heterogeneous artifacts. For this purpose, each language artifact is associated to its exact model type (e.g., L_1 to MT_1 and L_2 to MT_2 in Figure 6.1). Imported artifacts (metamodels and semantics) are also associated to their structural interface in the form of a model type (MT_3 and MT_4). We introduce later in this chapter how these structural interfaces are inferred.

Finally, when reusing fragments of syntax and semantics from one language to another, one would also like to reuse the tools and services associated to them. Several languages may *implement* the same model type, meaning that transformations and tools defined over a model type can be reused for all matching languages. The framework for model typing introduced in Chapter 5 enables reasoning about compatibility between different artifacts, e.g., to check whether a given semantics can be applied to a given syntax, or to ensure that within an inheritance relation the sub-language remains compatible with the tools and services of its super-languages.

6.3 An Algebra for DSL Assembly and Customization

In this section, we introduce an abstract algebraic specification of operators for language assembly and customization. Various implementation choices, such as the concrete formalisms for syntax and semantics definition of languages, are left open in the specification. This specification is mainly intended to serve as a reference for the implementation of a concrete meta-language that would support the aforementioned approach. It is up to the implementor to instantiate the abstract operators, e.g., in the context of a particular language workbench. Section 6.4 details how we instantiate the operators in the Melange language workbench, where Ecore is used for defining the abstract syntax of language, and K3 their semantics. We first provide the definitions and concepts required to define the algebra (Section 6.3.1). Then, we formalize the operators for language assembly (Section 6.3.2) and language customization (Section 6.3.3).

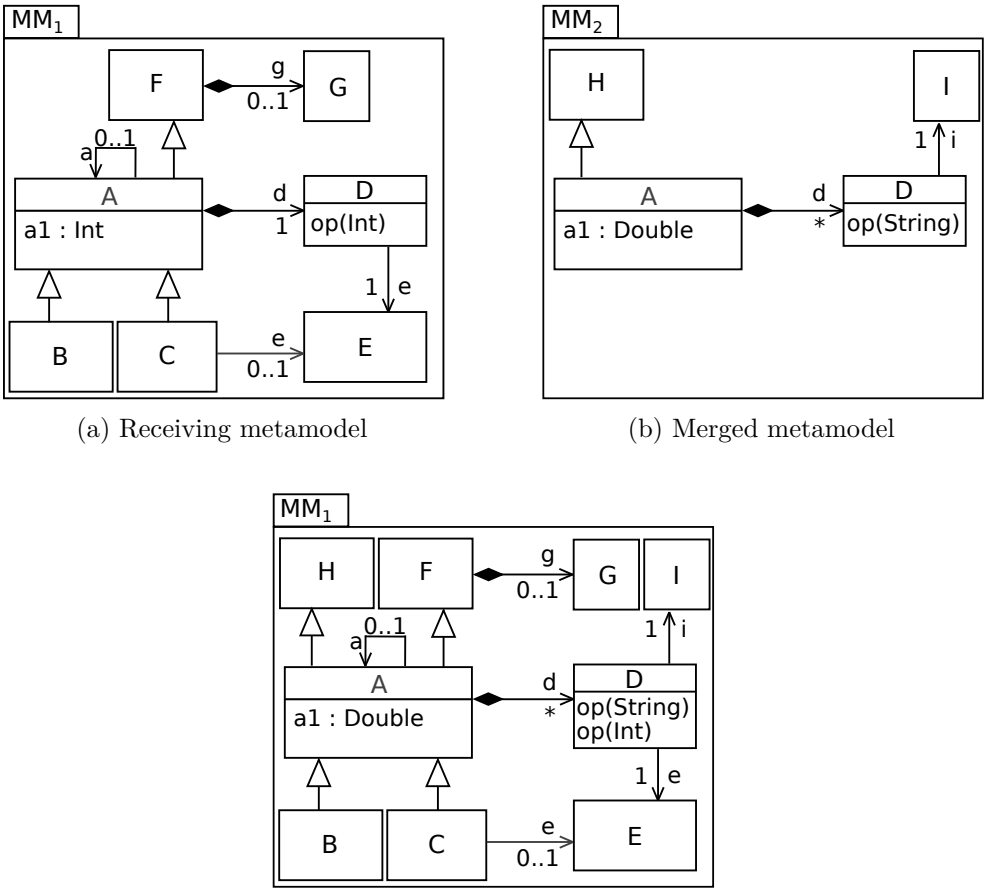
6.3.1 Language Definition

Based on the informal conceptual model of Section 6.2, we define a language $\mathcal{L}$ as a 3-tuple of its abstract syntax, semantics, and exact model type:

$$\mathcal{L} \triangleq \langle AS, Sem, MT \rangle \tag{6.1}$$

Including the exact model type of a language into the tuple allows to directly specify the impact of each of the operators of the algebra on the typing layer. Thereby, this makes explicit in which cases tools and services can be reused from one language to another, based on the model polymorphism mechanism introduced in Chapter 5. In the remainder of this section, for any language $\mathcal{L}$, we denote $AS(\mathcal{L})$ its abstract syntax, $Sem(\mathcal{L})$ its semantics, and $MT(\mathcal{L})$ its exact model type. On non-ambiguous cases, we simply refer to them as AS, Sem, and MT. The next subsections detail each of these constituents.

Syntax and Syntax Merging In our algebra, the abstract syntax AS of a language $\mathcal{L}$ is specified using a metamodel, i.e., a multigraph of meta-classes and their relations [121]. When assembling several abstract syntaxes, their concepts must be merged together so that the resulting abstract syntax is no less capable than its ancestors [90]. Informally, this means that the abstract syntax resulting from the merge must incorporate concepts from all languages and merge the definitions of shared elements. In our specific case, merging several abstract syntaxes boils down to the problem of metamodel composition [88]. Figure 6.2 illustrates the syntax merging operator on a simple example. In this example, the metamodel depicted in Figure 6.2b is merged into the metamodel



(a) Receiving metamodel

(b) Merged metamodel

(c) Resulting metamodel

Figure 6.2: The syntax merging operator

depicted in Figure 6.2a to produce the metamodel depicted in Figure 6.2c. We use the terms *receiving metamodel*, *merged metamodel*, and *resulting metamodel* to refer to the three metamodels involved in the merging operation. Similarly, the terms *receiving language*, *merged language*, and *resulting language* will be used throughout this section.

Depending on the meta-language used for defining metamodels, different merging operators may be employed with different policies for matching and merging rules, conflicts management, etc. The choice of the concrete semantics of the syntax merging operator is left to the implementer of the algebra, and a concrete implementation is described in Section 6.4.2. In the remainder of this section, we denote $\circ$ the abstract syntax merging operator.

Semantics and Semantics Merging The semantics *Sem* of a language $\mathcal{L}$ consists in a sequence of aspect definitions A_i^t, where A is a class, t is a

pointcut and i is the index of A in the sequence. In this case, the pointcut t specifies the concept of the language's abstract syntax (a meta-class) on which the aspect must be ultimately woven. This makes explicit the *binding* relation of Figure 6.1 between the syntax and the semantics of a language. The advice is the class A itself, consisting of attributes and methods. When a joinpoint is found, i.e., when a matching concept is found in the language, elements of the advice are inserted in the target meta-class. Typically, for semantics definition, the advice consists of a set of methods that describe the computation steps of the concept matched by t [147]. Since aspects are defined using classes in an object-oriented fashion, they may inherit from each other. To cope with possible specializations and redefinitions of methods, aspects are ordered by hierarchy in a sequence:

$$\begin{aligned} Sem(\mathcal{L}) \triangleq (A_i^t \in Aspects) \text{ where} \\ \forall A_i^t \in Sem(\mathcal{L}), \exists c \in AS(\mathcal{L}) : c <\# t \\ \forall A_i^t, A_j^t \in Sem(\mathcal{L}) : A_i^t \lhd A_j^t \implies i > j \end{aligned} \tag{6.2}$$

where $<\#$ denotes the object type matching relation introduced by Bruce and Vanderwaart [35] and used to formalize the model typing relation [239], and $\lhd$ denotes the class inheritance operator. For a language $\mathcal{L}$ to be well-formed, each of the aspects of its semantics $Sem(\mathcal{L})$ must have a matching meta-class in its abstract syntax $AS(\mathcal{L})$; this is what the first property ensures. Ordering the aspects that compose a semantics in a sequence leaves the choice of linearization and/or disambiguation opened to the implementer when several aspects are in conflicts (e.g., insert the same method on the same target t). The merging of two semantics, denoted $Sem \bullet Sem'$, consists in producing a new sequence of aspects. As the definition shows, merging two semantics is equivalent to concatenating their sequences of aspects. As a result, this operator is not commutative and any redefinition of an aspect or method in Sem' overrides the previous definition in Sem:

$$Sem \bullet Sem' \equiv Sem \frown Sem' \tag{6.3}$$

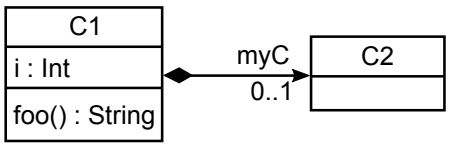
where $\frown$ denotes the sequence concatenation operator. We also denote $sig(A)$ the *signature* of an aspect A. The signature of an aspect is a metamodel that exposes all the features (i.e., properties and methods) defined in an aspect and its dependencies, omitting the concrete method bodies. Figure 6.3b depicts the signature extracted from the aspect depicted in Figure 6.3a using the K3 meta-language.

It follows that the signature of a semantic specification Sem is defined as the structural merge (through $\circ$) of the signature of the aspects that compose it:

$$sig(Sem) \triangleq \bigcup_{A_i^t \in Sem}^{\circ} sig(A_i^t) \tag{6.4}$$

```
1 @Aspect(className = C1)
2 class A {
3   int i
4   @Containment C2 myC
5
6   def String foo() {
7     [...]
8   }
9 }
```

(a) An aspect A expressed in K3



(b) The corresponding signature $sig(A)$

Figure 6.3: The signature $sig(A)$ of an aspect A

Model Typing Each language $\mathcal{L}$ has one exact model type MT. Like abstract syntaxes, model types are described with a metamodel. The exact model type of a language is its most precise structural interface, i.e., the model type that exposes all its features. Thus, the exact model type of a language exposes both its concepts and their relations (i.e., its metamodel) and the signature of its semantics (newly inserted features and methods). Hence, the exact type MT of a language $\mathcal{L}$ is defined as the structural merge of its abstract syntax and the signature of its semantics:

$$MT(\mathcal{L}) \triangleq AS(\mathcal{L}) \circ sig(Sem(\mathcal{L})) \tag{6.5}$$

Any change in either the abstract syntax or the signature of the semantics of a language will result in a different type. The issue of tooling is indirectly addressed through the reasoning capabilities provided by the model typing layer: if the result of the application of operators leads to a language $\mathcal{L}$ whose model type MT is a subtype of the model type MT' of another language $\mathcal{L}'$, then tools defined for $\mathcal{L}'$ can be reused as is for $\mathcal{L}$. In the following, we denote $<:$ the subtyping relation between model types.

6.3.2 Operators for Language Assembly

Syntax Merging When building new languages, it is likely that previously defined language abstract syntax fragments may be reused as is. For instance, the syntactic constructs of a simple action language (e.g., with expressions, object manipulation, basic I/O) may be shared by any language encompassing the expression of queries or actions. This first scenario of language assembly thus consists in importing a fragment of abstract syntax from another language to reuse its definition. In such a case, the language resulting from the merge of the receiving language and the merged abstract syntax must incorporate all the concepts of both, while preserving the semantics of the receiving language. Also, its model type must be updated accordingly to incorporate the new

syntactic constructs. Hence, we specify the merging of an abstract syntax AS into a language $\mathcal{L}$, denoted $\overset{m}{\leftarrow}$, as follows:

$$\mathcal{L} \overset{m}{\leftarrow} AS' = \langle AS \circ AS', Sem, MT \circ AS' \rangle \tag{6.6}$$

In most cases, the resulting model type $MT' = MT \circ AS'$ is a subtype of both AS' and MT since it incorporates the features of both. It is, however, worth noting that new elements introduced in a model type with the $\circ$ operator may break the compatibility with the super model type in some cases (e.g., the introduction of a new mandatory feature [119]). In the former case, when the compatibility can be ensured through subtyping, tooling defined over AS' and/or $\mathcal{L}$ (e.g., transformations, checkers) can be reused as is on the resulting language.

Semantics Weaving Another scenario of language assembly consists in importing predefined semantics elements in a language. When different languages share some close abstract syntax, such as different flavors of an action language, their semantics are likely to be similar, at least for the common subparts (e.g., the semantics of integer addition is likely to remain unchanged). When the case arises, one would like to import the semantics definition of addition from one action language to another. We denote $\overset{w}{\leftarrow}$ the semantics weaving operator, which consists in weaving a semantics Sem' on a language $\mathcal{L}$. In such a case, the two semantics are merged and the exact type of $\mathcal{L}$ is updated to incorporate the syntactic signature of the new semantics:

$$\mathcal{L} \overset{w}{\leftarrow} Sem' = \langle AS, Sem \bullet Sem', MT \circ sig(Sem') \rangle \tag{6.7}$$

Following the previous definitions, this operator can be successfully applied only if there is a matching meta-class in AS for each aspect of Sem'. That is, $\forall A_i^t \in Sem', \exists c \in AS : c <\# t$. Because the two semantics are concatenated through $\bullet$, Sem' may override any previous definition of Sem (such as a particular computation step implemented as a method in an aspect). It follows that the semantics weaving operator may be employed either to augment or to override part of the semantics of the receiving language $\mathcal{L}$. The semantics weaving operator is thus particularly relevant for incrementally implementing semantic variation points [44].

6.3.3 Operators for Language Customization

In the previous section, we specified how the syntax merging and semantics weaving operators help to build new languages by assembling predefined fragments of abstract syntax and semantics. However, although the reuse of language artifacts significantly decreases the development costs, the resulting languages may not fit exactly the language designer's expectations, or the particular requirements of a new domain. We introduce in this section an

algebra for language customization. Customization may include specialization of the abstract syntax or semantics of a language for a given context, restriction to a subset of its scope or composition with (possibly part of) other language definitions. In a recent paper, Erdweg et al. propose a taxonomy of different composition operators between languages, namely language extension, restriction, unification, and self-extension [90]. The operators of our algebra closely match their taxonomy: the *inheritance* operator supports language extension, the *slicing* operator supports language restriction, and the *merging* operator supports language unification. Self-extension is deemed out of scope since it refers to the ability of a language to extend itself (aka. language embedding). We detail each of our language customization operators in the next subsections.

Language Merging Situations arise where two independent languages must be composed to form a more powerful one. For instance, a finite-state machine language may be defined as a basic language of states and labeled transitions combined to an action language for expressing complex guards and actions. The resulting language may in turn be merged with a language for expressing classifiers where the state machines would describe the behavior of their methods. Another intuitive example is the association of markup languages (e.g., HTML) styling languages (e.g., CSS) and programming languages (e.g., Javascript), which together enable the development of modern dynamic webpages. Instead of keeping these languages separated – resulting in e.g., coherency problems – an alternative would be to unify them in a single, coherent, language for web engineering. To support this kind of scenario, we introduce the language merging operator, denoted $\uplus$. The output of this binary operator is a new language that incorporates the syntax and semantics of its two operands. In this case, the receiving language is augmented with the merged language to produce the resulting language. Because the merged language $\mathcal{L}'$ can override part of the semantics of the receiving language $\mathcal{L}$, the two languages do not commute under $\uplus$.

$$\mathcal{L} \uplus \mathcal{L}' = \langle AS \circ AS', Sem \bullet Sem', MT \circ MT' \rangle \tag{6.8}$$

Naturally, language merging may be equivalently reformulated in terms of language assembly operators:

$$\mathcal{L} \uplus \mathcal{L}' \equiv (\mathcal{L} \overset{m}{\leftarrow} AS') \overset{w}{\leftarrow} Sem' \tag{6.9}$$

The resulting language incorporates the syntax of both operands (merging the common concepts), as well as their semantics (with possible redefinitions in Sem'). The exact model type $MT'' = MT \circ MT'$ of the resulting language results from the merge of the exact types of the operands: it exposes the features of both. In most cases, the resulting exact type MT'' is thus a subtype of both MT and MT'. If one of the two languages exposes a mandatory property that is not present in the other, however, the subtyping relation does not stand [119].

Language Inheritance In essence, the language inheritance operator is similar to the language merging operator, as both aims to combine the definitions of two languages. The language inheritance operator, denoted $\oplus$, differs from the language merging operator in that it does not consider the two languages on equal terms: a *sub-language* inherits from a *super-language*. Moreover, the language inheritance operator ensures that the sub-language remains compatible with its super-language. Regardless the subsequent operators applied to the sub-language, it must remain compatible with the super-language, otherwise an error is reported. Concretely, it means that the exact model type of the sub-language must remain a subtype of the exact model type of the super-language: the $MT <: MT'$ property is conservative, meaning that any operators apply on $\mathcal{L}$ must not violate it. In a sense, the language inheritance operator supports a form of language design-by-contract, as the language designer is assured that tools defined over $\mathcal{L}'$ can be reused on $\mathcal{L}$.

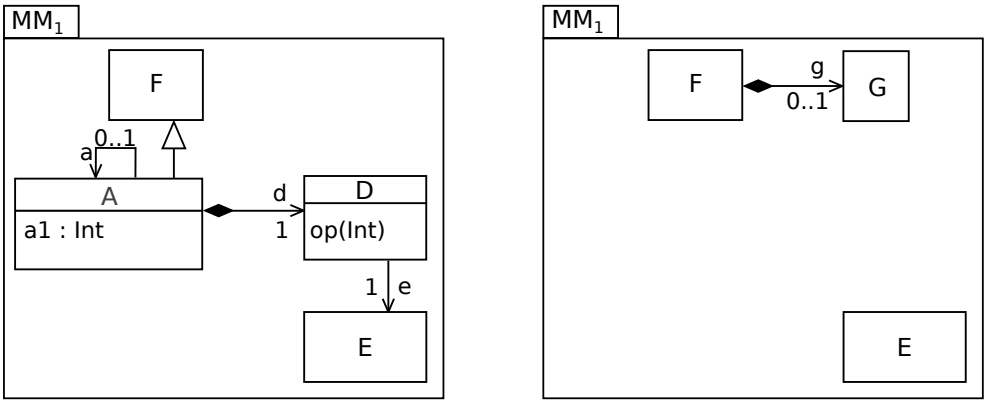
$$\begin{aligned} \mathcal{L} \oplus \mathcal{L}' = \langle AS \circ AS', Sem' \bullet Sem, MT'' \rangle \text{ where} \\ MT'' = MT \circ MT' \text{ and} \\ MT'' <: MT' \end{aligned} \tag{6.10}$$

Note that in this case, the sub-language first inherits the abstract syntax and semantics of its super-language, and may then override part of the inherited artifacts to refine its definition, provided that the conservative subtyping relation stands.

Naturally, the language inheritance operator can be equivalently formulated in terms of assembly operators, provided that the aforementioned constraints hold:

$$\mathcal{L} \oplus \mathcal{L}' \equiv (\mathcal{L}' \xleftarrow{m} AS) \xleftarrow{w} Sem \tag{6.11}$$

Language Slicing Model slicing [28, 234] is a model comprehension technique inspired by program slicing [272]. The process of model slicing involves *extracting* from an input model a subset of model elements that represent a *model slice*. Slicing criteria are model elements from the input model that provide entry points for producing a model slice. The slicing process starts by slicing the input model from model elements given as input (the slicing criteria). Then, each model element linked (e.g., by inheritance or reference) to a slicing criterion is sliced, and so on until no more model elements can be sliced. For instance, model slicing can be used to extract the static metamodel footprint MM' of a model operation defined over a metamodel MM, i.e., extracting the elements of MM used by the operation [143]. Model slicing can be positive or negative. Positive model slicing consists of slicing models according to structural criteria. These criteria are the required model elements from which the slice is built. For instance, based on the simple metamodels of Figure 6.2, one may want to slice the MM_1 metamodel using as slicing criterion



(a) Positive slicing on MM_1 with $A.a$ as input

(b) Negative slicing on MM_1 with D as input

Figure 6.4: The language slicing operator

the reference a of the class A. This slicing consists of statically extracting all the elements of MM_1 in relation with a (a included). The result of this slicing is depicted by Figure 6.4a: the class A that contains a is sliced; the super class of A (F) is sliced; the A's references with a lower cardinality greater than 0 are sliced (only the mandatory references and attributes are sliced); the target classes of these references (e.g., D) are sliced. This slicing process continues recursively until no more elements can be sliced. We extend the model slicing principles proposed by Blouin et al. [28] to support negative slicing. Negative slicing consists of considering the slicing criteria as model elements not to have in the slice. For instance, a negative slicing of MM_1 with the class D as slicing criterion produces the slice depicted by Figure 6.4b: a clone, that will be the output slice, of MM_1 is created; The class D is removed from this clone; all the classes that have a mandatory reference to D are removed (class A); subclasses of the removed classes are also removed (classes B and C). This slicing process continues recursively until no more elements can be removed.

Model slicing can be used to perform language restriction. For instance, a language designer may want to shrink a legacy metamodel to its sub-set used by a set of model operations of interest [143]. This consists of a positive slicing from a set of operations. A language designer may also want to restrict the features of a language (e.g., removing specific features of a programming language) for education purposes or to reduce its expressiveness [90]. This consists of a negative slicing from unwanted elements. A concrete example would be the extraction of one of the diagram of UML (e.g., the class diagram) to be reused in another context.

In the context of language engineering, we leverage the slicing operation to permit language designer to slice a language according to some slicing criteria, as formalized as follows. Given a language $\mathcal{L} \triangleq (AS, Sem, MT)$.

Slicing $\mathcal{L}$ using the slicing criteria c consists of slicing positively or negatively (respectively denoted Λ^+ and Λ^-, or Λ^+_- when considering both operators) its abstract syntax AS using c to produce a new abstract syntax AS', such that $AS' \subseteq AS$. Then, the aspects A^t_i, that compose Sem, that only refer to elements defined in AS' are extracted to form Sem', as formalized as follows:

$$\begin{aligned} \Lambda^+_-(\mathcal{L}, c) = \langle AS', Sem', MT' \rangle, \text{ where:} \\ AS' \triangleq \lambda^+_-(AS, c),\ AS' \subseteq AS, \\ Sem' \triangleq (A^t_i \in Sem, fp(A^t_i, AS) \subseteq AS'), \\ MT <: MT' \end{aligned} \tag{6.12}$$

The footprint operation (denoted fp) extracts the metamodel elements of AS used in the aspects a, similarly to model operation footprinting [143]. The choice of applying a positive ($\Lambda+$) or negative ($\Lambda-$) slicing is made by the language designer during at language design time according to her requirements. The abstract syntax slicing operation [28] (denoted λ^+, λ^-, or λ^+_-) slices a given abstract syntax AS according to slicing criteria c to produce an output abstract syntax AS'. Because of the strict slicing that extracts metamodel elements by assuring the conformance, the original model type MT is a subtype of the resulting model type MT'.

6.4 Implementation of the Algebra in Melange

After introducing Melange's support for model polymorphism in Chapter 5, we detail in this section the features of Melange that are targeted to the implementation of the algebra specified in Section 6.3. Specifically, we detail the implementation choices used to instantiate the algebraic specification into the Melange language. We refer the reader to Chapter 7 for an in-depth presentation of Melange and K3.

Instead of providing its own dedicated meta-languages for the specification of each part of a DSL (abstract syntax, type system, semantics, etc.), Melange relies on other independently-developed components to provide such features. The abstract syntax of DSLs is specified using the Ecore implementation of the EMOF standard provided by the Eclipse Modeling Framework (EMF).[2] The choice of Ecore is motivated by the success of EMF both in the industry and academic areas. This allows Melange to possibly integrate a wide range of existing DSLs: over 300 Ecore metamodels exists in the "metamodel zoo" [246], over 9000 on Github. For semantics specification, Melange relies on the K3 meta-language to express operational semantics through the definition of aspects. The algebra introduced in Section 6.3 has been implemented within Melange, providing features for assembly and customization of legacy DSLs artifacts. Overall, Melange is tightly integrated with the EMF ecosystem.

[2] https://www.eclipse.org/modeling/emf/

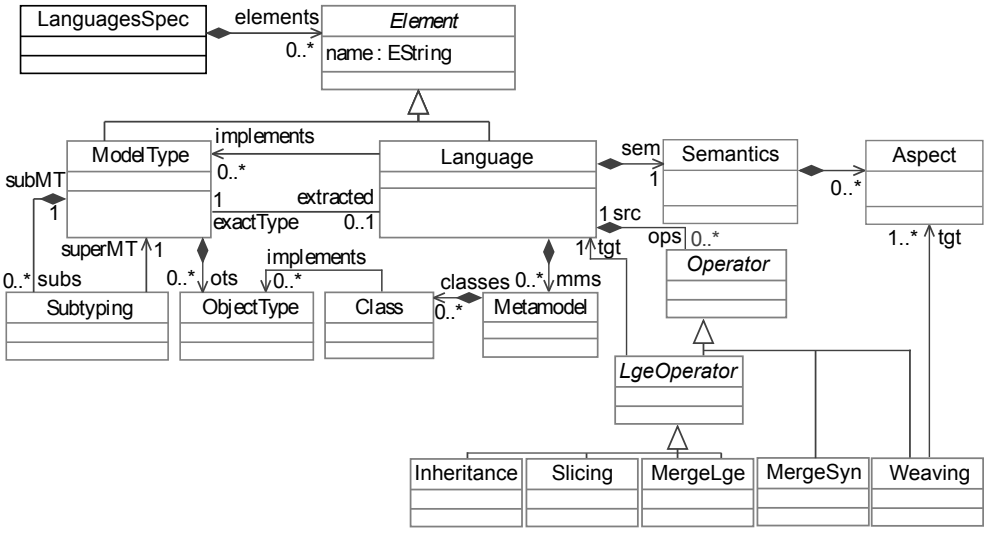


Figure 6.5: Excerpt of the abstract syntax of Melange

Newly built DSLs can thus benefit from other EMF-based components such as Xtext [93] for defining their textual syntax or Sirius [85] for their graphical representation.

In this section, we present the Melange language through its abstract syntax (Section 6.4.1), implementation choices (Section 6.4.2), and integration with the EMF ecosystem (Section 6.4.3). The concrete syntax and pragmatics of Melange are exemplified through the case study presented in Section 6.5.

6.4.1 Abstract Syntax

The abstract syntax of Melange (the metamodel depicted in Figure 6.5) includes the concepts and relations discussed in Section 6.2. *LanguagesSpec*, the root of Melange's abstract syntax, defines a meta-program that (i) specifies an assembly of DSLs (ii) delimits the scope for the inference and checking of model typing relations.

A *Language* is defined by its *Metamodel* and its associated *Semantics*. A *Metamodel* is composed of a set of *Class*es. A *Semantics* consists of a set of *Aspect*s used to weave behavior into its meta-classes. This mechanism relies on static introduction and is inspired by the concept of open classes [147]. As specified in Section 6.3, both assembly and customization operators can be applied on languages. The *MergeSyn* operator is used to import and merge a metamodel into a given language, whereas the *Weaving* operator is used to weave a given aspect on the abstract syntax of a language. For the customization part, a language can inherit (*Inheritance* operator) from a "super" language. The *MergeLge* operator allows language designers to merge

one language into another one. Finally, the *Slicing* operator permits to slice a language given a specific slicing criterion.

A *ModelType* defines an interface to manipulate models. It consists of a set of *ObjectType*s, thereby defining a group of interrelated types. Model types can be created from scratch, or automatically inferred from a concrete language. In the latter case, the language explicitly references this new model type as its *exactType*. Model types are linked one another by subtyping relations: if MT' is a subtype of MT, then there is one and only one *Subtyping* instance that references MT' as its *subType* and MT as its *superType*.

6.4.2 Implementation Choices

The algebra introduced in Section 6.3 can be implemented in various ways. We report here on additional choices we made in its concrete implementation within Melange. The algebra does not impose a particular formalism for expressing metamodels. In our implementation, we rely on the Ecore implementation of the EMOF standard provided by EMF to specify the abstract syntax of DSLs. Different operators for metamodel merging have been proposed in the literature (e.g., [163, 88]). As introduced in Chapter 3, the UML2.0 specification introduces the notion of *PackageMerge* that specifies "how the contents of one package are extended by the contents of another package" [207]. Informally, the UML specification states that "a resulting element will not be any less capable than it was prior to the merge". Matching of elements of both sides mostly occurs based on name equality. When a match is found between two elements, the resulting package incorporates both sides of its definition. We choose to use a slightly improved version of the *PackageMerge* operator as defined in the UML specification and refined by Dingel et al. [82]. To meet our requirements, we adapt the *PackageMerge* operator by trading its UML specificities with EMOF specificities, while preserving its general spirit. For example, we do not consider the concept of *Profile* and adapt the concept of UML *Association* to the concept of EMOF *Reference*. The *PackageMerge* operator specifies a set of constraints that must be ensured for the merge to succeed. If one of the constraints is violated, the merge is ill-formed and an error is reported. It follows that operators of the algebra that rely on the abstract syntax merging operator share the same property: if the constraints are violated the operation is invalid and an error is reported to the user, otherwise the result is guaranteed to be well-formed. On the semantics part, we choose to use the K3 meta-language, based on the Xtend programming language, for the definition of operational semantics. Xtend compiles directly to Java code, providing a seamless integration with other artifacts generated using the EMF framework. A simple example of an aspect used to weave executability in the *State* meta-class of a FSM language is given in Listing 6.1. The special variable `_self` refers to the element on which the aspect is ultimately woven, i.e., its joinpoint (the `fsm.State` meta-class in this

case). It allows the aspect to access all its features (e.g., `outgoingTransitions` in Listing 6.1). Here, the `ExecutableState` aspect inserts a `step` method in the `State` meta-class to fire the appropriate transition given an input character `c`. Note that aspects may also declare new attributes and references that are introduced in the target meta-classes.

```
1 @Aspect(className = fsm.State)
2 class ExecutableState {
3   def void step(char c) {
4     val t = _self.outgoingTransitions
5             .findFirst[input == c]
6     if (t == null) throw new Exception
7     else t.fire
8   }
9 }
```

Listing 6.1: Weaving executability with aspects

The `@Aspect` annotation specifies the pointcut of the aspect, while the rest of the class definition defines its advice (new methods and attributes to be inserted). Since pointcuts and advices are not explicitly separated, the process of re-binding a set of aspects to a new abstract syntax consists in copying the aspects while updating their pointcuts to target the appropriate concepts of the new abstract syntax. Note that in Listing 6.1, the `step` method of `ExecutableState` assumes the presence of a `fire` method in `fsm.Transition`. This method could be directly defined in the metamodel of the FSM language or, most likely, inserted using another aspect `ExecutableTransition`.

We also made the following choices in the priorities given to each operator. The inheritance operator has the highest priority, followed by the merge and slice operator (in order of appearance), ending with the aspect weaving operator. The rationale behind these choices is as follows. First, languages may inherit part of their definition from a super-language. As a consequence, the type system ensures that the sub-typing relation between the two languages is kept, otherwise an error is reported. Then, other artifacts may be assembled, merged or sliced on top of the inherited definition. Finally, aspect weaving comes last to support both the redefinition of imported parts and the addition of "glue code" to make the different parts fit together. As an example, when two merged languages exhibit no common subparts, a new aspect can be woven to connect them in a meaningful way by adding structural references between their abstract syntax, or by inserting some additional code to make their respective interpreters cooperate, e.g., through context translation. Finally, for each language declaration, Melange infers its corresponding exact model type. The embedded model-oriented type system automatically infers the subtyping hierarchy through structural typing. This hierarchy is used to ensure the subtyping relation when inheritance is involved, as described in Chapter 5.

6.4.3 Compilation Scheme and Integration with EMF

From a Melange specification, the Melange compiler first reads and imports the external definitions and assembles them according to the rules of the algebra. Once the new DSLs are assembled, customization operators are applied. Then, the compiler completes the resulting model by inferring the subtyping hierarchy among the model types inferred for each language. The implementation relations between metamodels and model types are also inferred in this phase, leading to a complete Melange model conforming to the metamodel of Figure 6.5. Then, it generates a set of artifacts for each declared language: (i) an Ecore file describing its abstract syntax (ii) a set of aspects describing its semantics attached to the concepts of its abstract syntax (iii) an Ecore file describing its exact model type and (iv) an Eclipse plug-in that can be deployed as is in a new Eclipse instance to support the creation and manipulation of models conforming to it. To generate the runtime code for the new artifacts, Melange relies on the EMF compiler (a *genmodel* generating Java code from an Ecore file), and the Xtend compiler (generating Java code from the aspects file). For each language definition, the Java code generated by both compilers is associated to a plug-in. Since Melange reuse the formalism for language definition of EMF, along with its compilation chain, it is fully interoperable with the EMF ecosystem. Newly created DSLs may thus benefit from other tools of the EMF ecosystem such as Xtext for the definition of a textual editor or Sirius for a graphical representation. More information on the compilation scheme of Melange and its integration with the EMF ecosystem is given in Chapter 7.

6.5 Case Study

In this section, we illustrate how the proposed algebra implemented within Melange can be used by language designers to leverage legacy DSLs in the creation of new ones. We introduce in Section 6.5.1 the case study we consider. Section 6.5.2 details how the proposed case study is implemented in Melange. Finally, Section 6.5.3 discusses the results we gather from this experiment.

6.5.1 Language Requirements

To illustrate Melange, we propose to design a new executable modeling language for the Internet of Things (IoT) domain, i.e., for embedded and distributed systems. This language is inspired by general-purpose executable modeling languages (e.g., Executable UML [182], fUML [233]) and modeling languages dedicated to the IoT domain (e.g., ThingML [103]). This language enables the modeling of the behavior of communicating sensors built on top of resource-constrained embedded systems, such as low-power sensors and micro-controller

devices (e.g., Arduino,[3] Raspberry Pi[4]). Such a language aims at providing appropriate abstractions and dedicated simulators, interpreters, or compilers depending on the targeted platforms. To illustrate the benefits of Melange, the resulting language is built as an assembly of other publicly-available languages. Instead of starting the definition of our language from scratch, we leverage the assembly and customization operators introduced in Section 6.3 to reuse as much as possible the definition of other languages developed externally. We consider the three following requirements while designing this language:

1. *The language has to provide an IDL (Interface Definition Language) to model the sensors' interfaces in terms of provided services.* Examples of popular languages that provide the appropriate abstractions include the class diagram of (f)UML, the SysML block definition diagram [210], MOF, or languages dedicated to interface definition such as the OMG IDL [203], as they all provide an object-oriented interface definition language.

2. *The language must support the modeling of IoT scenarios.* Scenarios, or "sketches", model how different sensors and actuators hosted on different board interact with each other (e.g., a thermometer hosted on one board sends the current temperature to another board which in turn manages the heater accordingly). Various languages may be employed to model this concern. For instance, process modeling languages such as the (f)UML/SysML activity diagram, BPEL [4], or BPMN [205] are good candidates.

3. *The primitive actions that can be invoked within the activities must be expressed with a popular language IoT developers are familiar with.* Such a language can be shared by the community and embedded on a set of devices used in the IoT domain. Even though the C language is the common base language of most embedded platforms, its lack of abstraction hinders its exploitation in a modeling environment. Instead, we choose the Lua language [140]. Lua is a dynamically-typed language commonly used as an extension or scripting language. Lua is notably popular in the IoT domain since it is compact enough to fit on a variety of host platforms.

The first step of the design of the IoT language is to define its minimal abstract syntax which captures its domain. Figure 6.6 depicts the initial metamodel of the IoT language. A `System` is composed of a set of boards (e.g., an Arduino) which host a set of hardware components (e.g., a thermometer, a motor). Each `HWComponent` exposes a set of services that are described by

[3] http://www.arduino.cc/
[4] https://www.raspberrypi.org/

an `OperationDef`, i.e., a method with a name, a list of typed parameters and a return type. Finally, the scenario one would like to experiment on the system is called a `Sketch` which is described by an `Activity` diagram. It is important to note that `OperationDef` and `Activity` are merely placeholders. Leveraging the assembly and customization operators proposed in Section 6.3, the concrete implementation of these concepts will be provided by other languages that are suitable for describing services and activities. With the aim of validating Melange, the experimental protocol consists in selecting three publicly-available implementations of existing EMF-based languages to support the three aforementioned requirements.

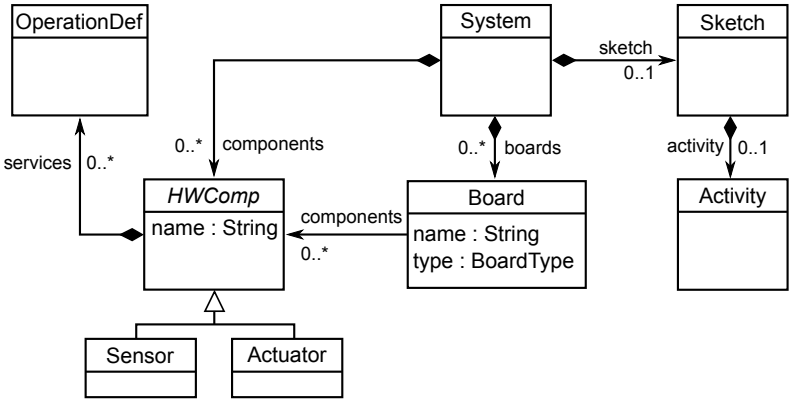


Figure 6.6: Initial metamodel of the IoT language

Interface Description Language For the definition of the structural interfaces of sensors, we choose the OMG Interface Description Language (IDL) [203], part of the Common Object Request Broker Architecture (CORBA) standard defined by the OMG [208]. The metamodel of IDL is depicted in Figure 6.7. Informally, the IDL enables the expression of user-defined data structures and interfaces, which consist in a set of attributes and operations. In the context of our IoT language, we use the IDL to express the structural interfaces of sensors in terms of provided services (e.g., to query the temperature in a room or turn on a light). We use an EMF-based implementation of the IDL language publicly available on Github.[5] It consists of an Ecore metamodel matching the abstract syntax described in the OMG specifications, and an Xtext grammar that eases the definition of textual models conforming to it. As we only use the IDL to express the structural interfaces of sensors, we do not require its semantics definition.

[5] https://github.com/catedrasaes-umu/idl4emf

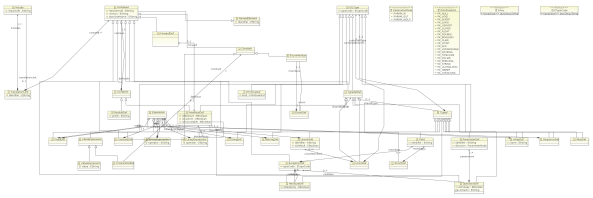


Figure 6.7: Excerpt of the metamodel of the IDL

Activity Modeling For the activity modeling part, we reuse materials from the *Model Execution Case* of the TTC'15 tool contest.[6] The case foresees the specification of the operational semantics of a subset of the UML activity diagram with transformation languages [180]. We directly reuse the Ecore metamodel specifying the abstract syntax of the activity diagram provided by the organizers.[7] For the operational semantics, we reuse the materials provided along our solution to the TTC'15 contest [61]. The operational semantics consists of a set of aspects expressed in K3, is publicly available on the companion webpage of our solution,[8] and detailed in our submission [61]. Figure 6.8 depicts an excerpt of the metamodel of the activity diagram.

Action Language For the action language part, we reuse a publicly-available implementation of the Lua language developed using Xtext.[9] We implement the operational semantics of the Lua language using K3 in the same way we defined the operational semantics of the activity diagram. Figure 6.9 depicts an excerpt of the metamodel of the Lua language.

6.5.2 Language Design using Melange

The resulting language is built using the Melange assembly definition given in Listing 6.2. We detail hereafter each part of the Melange file.

First, The abstract syntax of the IDL language is imported using the `syntax` keyword to define a new language named `Idl` (Lines 3–5). Then, the `ActivityDiagram` language is defined in the same way but uses the `with` keyword to import the aspects defining its operational semantics (Lines 7–11). The wildcard import `org.activitydiagram.semantics.*` specifies that all the aspects contained in the `org.activitydiagram.semantics` must be imported. Additionally, the definition of the `ActivityDiagram` languages involves the merge of another metamodel (Line 9) that inserts the runtime concepts necessary to express the semantics. In this case, the runtime concepts consist

[6] http://www.transformation-tool-contest.eu/

[7] https://code.google.com/archive/a/eclipselabs.org/p/moliz/source/ttc2015/source

[8] http://gemoc.org/ttc15/

[9] https://xtexterience.wordpress.com/2011/05/17/xtext-based-lua-editor/

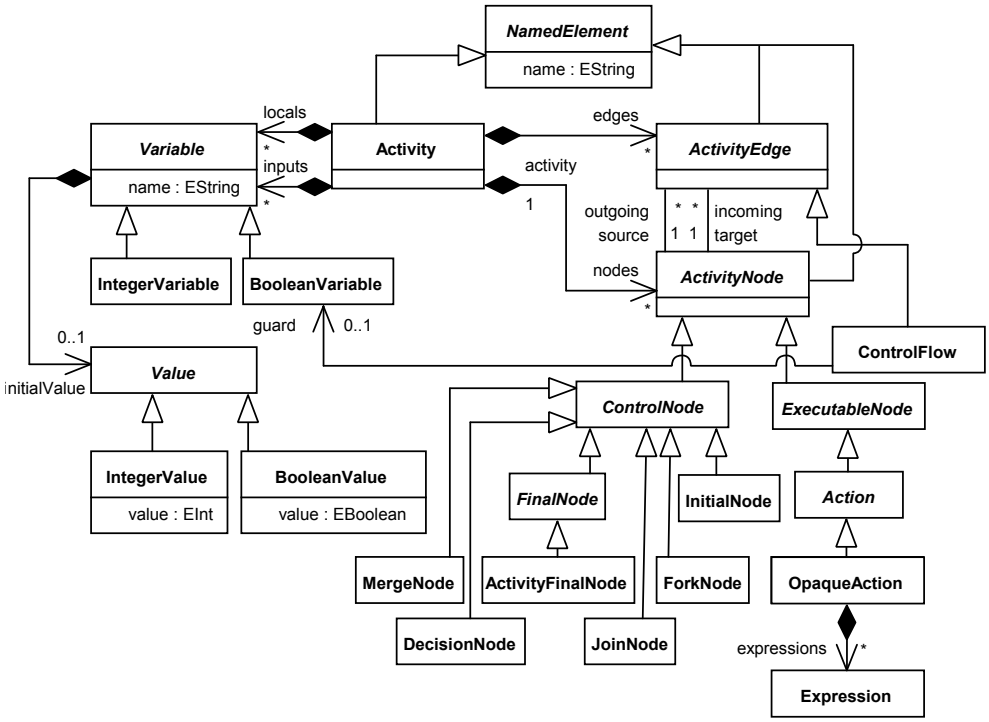


Figure 6.8: Excerpt of the metamodel of activity diagram (from [180])

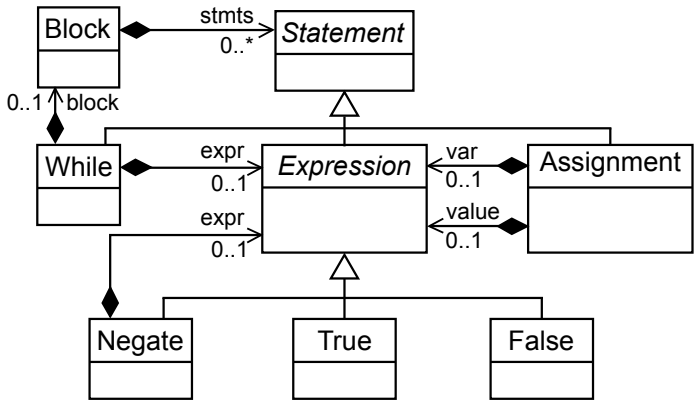


Figure 6.9: Excerpt of the metamodel of Lua

of *Tokens* and *Offers*. The interested reader can refer to the work of Syriani and Ergin for a formalisation of the semantics of activity diagrams in terms of tokens and offers as Petri nets [243]. The `Lua` language is defined in the same way as the assembly of its abstract syntax and operational semantics (Lines 13–16).

Finally, the `IoT` language is defined as an assembly of the three previously-defined languages. First, its **`syntax`**, depicted in Figure 6.6, is imported. Then,

its specification is merged with the result of the **slice** operation (Lines 20–21). This slice specifies that only the `OperationDef` and `PrimitiveDef` of `Idl` – and their mandatory dependencies – must be retained in the resulting language. All other concepts of `Idl` – which are not relevant for our IoT language – are discarded. Then, the resulting language is merged with the `Lua` language (Line 22) and `ActivityDiagram` language (Line 24). For the sake of clarity, the **renaming** clauses ensure that all the concepts imported from other languages using the **slice** and **merge** operators end up in the same package `iot` (Lines 21, 23, 25).

```
1  package fr.inria.diverse.iot
2
3  language Idl {
4    syntax "IdlMM.ecore"
5  }
6
7  language ActivityDiagram {
8    syntax "ActivityDiagram.ecore"
9    syntax "RuntimeConcepts.ecore"
10   with org.activitydiagram.semantics.*
11 }
12
13 language Lua {
14   syntax "Lua.ecore"
15   with org.lua.semantics.*
16 }
17
18 language IoT {
19   syntax "IoT.ecore"
20   slice Idl on ["OperationDef", "PrimitiveDef"]
21     renaming { "idlmm" to "iot" }
22   merge Lua
23     renaming { "lua" to "iot" }
24   merge ActivityDiagram
25     renaming { "activitydiagram" to "iot" }
26   with fr.inria.diverse.iot.OpaqueActionAspect
27   with fr.inria.diverse.iot.OperationDefAspect
28 }
```

Listing 6.2: Assembling the IoT language with Melange

At this point, because there is no overlapping concepts between the assembled languages (i.e., concepts with the same name), the new IoT language consists of concepts that are not yet linked together. To glue together these concepts and their semantics, two new aspects are woven on the assembly (Lines 26–27). The role of these aspects is to create links between the assembled languages, and customize them to fit the IoT specificities.

Listing 6.3 depicts the `OperationDefAspect`. This aspect glues together the `OperationDef` meta-class of the `Idl` language and the `Block` meta-class of the `Lua` language. On the syntactic side, it inserts a new containment reference from `OperationDef` to `Block`, specifying that each `OperationDef` service is implemented by a `Block` of Lua code. On the semantics side, it weaves a new method `execute` in the `OperationDef` meta-class that delegates its execution to the already-implemented `execute` method of `Lua`'s `Block` semantics. Similarly, the other aspect `OpaqueActionAspect` takes care of gluing together `ActivityDiagram`'s `OpaqueAction` with `Idl`'s `OperationDef`, specifying that an opaque action of the activity diagram can invoke a particular service on a particular board. For the sake of conciseness, the complete code of `OpaqueActionAspect` is given in Listing A.2.

```
1 @Aspect(className = OperationDef)
2 class OperationDefAspect {
3   @Containment
4   public Block lua
5
6   def void execute(Environment env) {
7     _self.lua.execute(env)
8   }
9 }
```

Listing 6.3: The `OperationDefAspect` linking IDL's `OperationDef` to Lua's `Block`

Each language is implicitly associated to its automatically-inferred exact model type (named after the language's name suffixed with MT, e.g., `LuaMT`). The type checking algorithm of Melange can thus infer the subtyping hierarchy among the different languages, as described in Chapter 5. In this case, for instance, the exact model type of the resulting language `IoTMT` subtypes both `LuaMT` and `ActivityDiagramMT`, as it exposes all of their features that have been retrieved through the **`merge`** operator. Consequently, tools and transformation defined on e.g., the `Lua` language can be reused to manipulate models conforming to the `IoT` language, thanks to model polymorphism. `IoTMT` does not subtype `IdlMT`, however, as the **`slice`** operator has removed all the concepts of `Idl` that are not related to `OperationDef` and `PrimitiveDef`. In the end, we obtain a new executable modeling language for IoT resulting from the composition of three legacy languages for which reuse was unforeseen. Additionally, the tools previously defined on the assembled languages can be reused as is for the resulting language.

As explained in Section 6.4.3, new languages built using Melange (such as the `IoT` language) are fully EMF-compliant. They can thus benefit from other tools of the EMF ecosystem. For the sake of the experiment, we designed both a textual editor (using Xtext) and a graphical editor (using Sirius) for our IoT

language. Listing 6.4 depicts an example model expressed in a textual form. The comments highlight which languages are used to write each part of the model.

```
1  system MySystem {
2    actuator Heater {
3      // IDL
4      provides operation lowerTemp(inout long temp) {
5        // Lua
6        temp = temp - 5
7      }
8    }
9    board B1 [Arduino] {
10     provides Heater
11   }
12   sketch {
13     // ActivityDiagram
14     activity IoTSketch {
15       int temp = 0,
16       [...]
17
18       nodes {
19         initial init out (e1),
20         merge m1 in(e1, e19) out (e2),
21         fork f1 in(e2) out (e3, e4),
22         action a1 comp { temp >= tempThreshold [...] }
23           in (e3) out (e5) service lowerTemp,
24         [...]
25       }
26       edges {
27         flow e1 from init to m1,
28         [...]
29       }
30     }
31   }
32 }
```

Listing 6.4: Example IoT model in Xtext

Figure 6.10 depicts the same model in its graphical form. The left part depicts the general sketch of the IoT scenario one would like to experiment in the form of an activity diagram. Each activity in the diagram corresponds to the invocation of a particular service. Services are represented on the right part of Figure 6.10 and are associated to a particular device (Arduino, Raspberry Pi, and Beagle boards in this case). The description of each service takes the form of a method, with optional parameters and return types, expressed using the IDL. Finally, the implementation of each of these services is specified in Lua (not shown in Figure 6.10). As the operational semantics of the IoT language

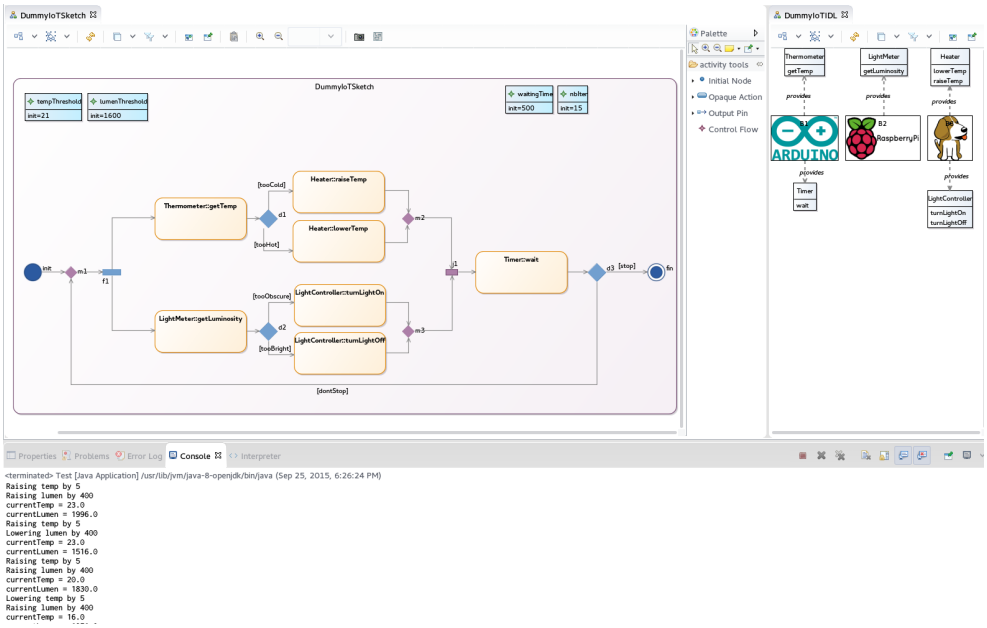


Figure 6.10: Executing a model conforming to the IoT language

is fully defined, this model can be executed – the output of its execution is shown in the lower part of Figure 6.10.

6.5.3 Results and Discussion

Ability of Melange to be integrated into an existing ecosystem – The integration using Melange of three existing EMF languages allows a language designer to obtain a new EMF language. If we do not consider the imposed methodology for defining the language semantics (the use of the interpreter pattern [110]), no modification of these languages was required to support that composition. This illustrates how Melange can be integrated into an existing language workbench without any change in the legacy abstract syntaxes. All the Melange operators are used for this case study. Although this does not guarantee that these operators are sufficient, it highlights that all of them are required when a language designer needs to compose existing languages.

Possible overhead in term of performance and lines of code that stem from the use of Melange – Compared to a top-down approach where the IoT language is built from scratch by an expert in language design, we observe no additional concepts integrated into the abstract syntax definition. At the semantics level, glue code is injected for the implicit conversion of the interpreter pattern context resulting from the composition of the various contexts stemming from various operational semantics. At runtime, no additional cost in terms of performance were observed to the use of the language resulting from the composition. Table 6.1 sums up the results.

	Melange	Top-down
Metaclasses (#)	104	104
LoC for the glue (#)	27	0
Efficiency (sec)	30,0	25,9

Table 6.1: Comparison of Melange and a top-down approach for the IoT language

Performance comparison is obtained by loading and executing a model with 10 objects that contains one operation with a workflow with 1000 basic actions that do mainly 10 numeric operations. The comparison was done on the same laptop designed with an Intel i7 with 16Gb of memory, a Linux 64bit operating system, and an Oracle Java 8 virtual machine.

Alignment with the taxonomy of language composition – Interestingly, our approach for assembling the three languages to create the IoT language perfectly matches Erdweg et al.'s taxonomy of language composition. In the experiment, we use language extension, restriction, unification, and composition extension. Moreover the composition is realized "by adding glue code only" [90], using aspects.

Nevertheless, these results may be moderated by the following threats to validity. First, all the languages must be designed in the same technical ecosystem. Melange does not provide any support for integrating heterogeneous languages in terms of technical ecosystem. Second, Melange cannot compose any language semantics. The composition can be done if and only if the semantics is operational and defined following the interpreter pattern (e.g., through static introduction or a visitor). Third, concepts with different names in different languages may represent the same concept. In such a case, adaptation mechanisms are required to align them before composition. Melange provides a simple renaming mechanism that allows to rename concepts, but lacks a powerful mechanism for realizing complex adaptations. Finally, the same person implemented the language using Melange and a traditional top-down approach. This person is an expert in language design and modeling technologies. Besides, the top-down language design has been reviewed by three experts in language design.

6.6 Conclusion

While current language workbenches provide import mechanisms, they usually lack an explicit support for customization and safe composition of imported artifacts. In this chapter, we proposed an approach for building DSLs by safely assembling and customizing legacy DSLs artifacts. We proposed different oper-

ators for assembling (merge_S/*weave*), restricting (*slice*), extending (*inherits*), and merging (merge_l) DSLs. The use of typing and subtyping relations that provides a reasoning layer for DSLs manipulation is also promoted. The approach is implemented in the Melange language workbench. We illustrated and discussed the usefulness of Melange by designing a new executable modeling language for IoT showing that: all the proposed operators are relevant for designing a new language based on the composition and the specialization of three legacy DSLs; the use of Melange does not introduce specific technical issue compared to a traditional top-down approach. Besides, our approach for assembling the three languages to create the IoT language perfectly matches Erdweg et al.'s taxonomy of language composition.

Part III

Implementation

Chapter 7

The Melange Language Workbench

In Chapters 5 and 6, I detail the specific features of Melange that support, respectively, model polymorphism, and modular language development. In this chapter, I give a broader description of the Melange workbench including its support for language definition, extension, and composition (Section 7.1), model typing (Section 7.2), and its integration with the EMF ecosystem (Section 7.3). Since Melange relies on the K3 meta-language for the definition of operational semantics, I also introduce K3 in greater details.

Melange [83] is a language workbench bundled as a set of Eclipse plug-ins, built atop the Eclipse Modeling Framework (EMF), and distributed under the Eclipse Public License (EPL-1.0). Melange integrates all the contributions presented in this thesis. We paid particular attention to its development in order to validate and spread our ideas in collaborative projects. In Chapters 5 and 6, we present the support in Melange for, respectively, model polymorphism and modular development of DSLs, including the implementation choices and technical details. This chapter is mainly intended as a reference manual that describes the different features of Melange from a user point of view. As a complement to this chapter, we refer the reader to the official websites of Melange[1] and K3[2] for the most up-to-date documentation.

7.1 Language Definition in Melange

Melange permits language designers to specify the abstract syntax and the operational semantics of software languages. Melange does not directly support the manipulation of concrete syntax specifications as first-class entities: it is thus not possible to compose two arbitrary grammars or graphical editors.

[1] http://melange-lang.org
[2] http://diverse-project.github.io/k3/

However, as we shall see, the languages generated by Melange are plain EMF languages. Language designers can thus manually define the concrete syntaxes of the generated languages, using other tools of the EMF ecosystem such as Xtext [93] or Sirius [85]. In this section, we present the two meta-languages employed to define abstract syntaxes and operational semantics. We use a simple extension of the `MiniFsm` language presented in Chapter 2 to illustrate the features of Melange. The complete source code of the example is available on the Melange repository.[3]

7.1.1 Abstract Syntax definition in Ecore

As introduced in Chapter 2, Ecore is an object-oriented meta-language for the specification of metamodels aligned with the EMOF specification of the OMG and bundled as part of EMF. In Melange, Ecore is used to specify the abstract syntax of languages in the form of a metamodel. Ecore being a popular language for metamodel definition (over 300 Ecore metamodels in the "metamodel zoo" and over 9000 on Github), new languages in Melange can leverage a rich variety of existing metamodels. As we have already introduced Ecore in Chapter 2 and used it in Chapters 5 and 6, we do not detail it further here. Figure 7.1 depicts the metamodel of the `MiniFsm` language we use throughout this chapter.

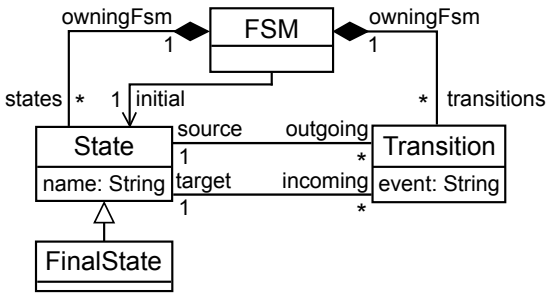


Figure 7.1: `MiniFsm`'s metamodel

7.1.2 Operational Semantics definition in K3

K3 is a meta-language for operational semantics definition inspired from Kermeta [147]. In particular, it follows an aspect-oriented modeling approach [146] for defining operational semantics where the computation steps of the semantics are implemented as methods contained in an aspect [147]. K3 aspects leverage the concepts of open-classes [57] and inter-type declarations [5] to statically weave new methods, attributes, and references in the corresponding meta-classes of a metamodel. Concretely, K3 consists of a set of active annotations

[3] https://github.com/diverse-project/melange/tree/master/examples/MiniFSM

on top of the Xtend programming language.[4] These annotations are processed by dedicated annotation processors that complement the Xtend compiler to generate the corresponding Java code at design time.

```
1  @Aspect(className = FSM)
2  class FSMAspect {
3    State currentState
4    String currentEvent
5
6    def void execute(EList<String> events) {
7      val eventIt = events.iterator
8
9      if(eventIt.hasNext)
10       _self.currentEvent = eventIt.next
11
12     _self.currentState = _self.initialState
13
14     while (_self.currentState !== null) {
15       _self.currentState.execute
16
17       if(_self.currentState instanceof FinalState)
18         _self.currentState = null
19       else {
20         val candidate = _self.transitions.findFirst[
21           input === _self.currentState && isFireable]
22         _self.currentState = candidate?.output
23       }
24     }
25   }
26 }
```

Listing 7.1: Weaving an `execute` method in `FSM` using K3

Listing 7.1 depicts a simple aspect inserting a new reference `currentState`, a new attribute `currentEvent`, and a new method `execute()` in the `FSM` meta-class depicted in Figure 7.1. The `@Aspect` annotation takes as parameter a `className` that points to an existing meta-class. The new *runtime concept* `currentState` stores a reference to the current `State` at each execution step, while the `currentEvent` attribute stores the event to be processed at a given step. In the context of an aspect, the `_self` variable is an implicit variable referring to the joinpoint of the aspect (a particular `FSM` instance). The `execute` method receives a list of events and processes them sequentially while triggering the execution of each state and firing the appropriate transitions. The methods `execute` on `State` and `isFireable` on `Transition` are themselves inserted by other aspects woven on the corresponding concepts, not shown here for the

[4] http://xtend-lang.org

sake of conciseness. Aspect methods call each other to define the overall control flow of the operational semantics, mimicking a classical visitor pattern [110].

Aspects can also inherit one another. This mechanism is particularly useful for customizing the semantics of an existing language, and gradually inserting semantic variation points. For instance, when extending a language, a new aspect can be woven on the resulting language to override one or several methods of its semantics (using the `@OverrideAspectMethod` annotation), possibly calling the super-implementation within the overriding method. Listing 7.2 depicts a simple aspect `FSMAspectOverride` that inherits from the `FSMAspect` of Listing 7.1 and overrides its `execute` method to first preprocess the input events, and then call back the overridden implementation of `execute` (Line 8).

```
1  @Aspect(className = FSM)
2  class FSMAspectOverride extends FSMAspect {
3    @OverrideAspectMethod
4    def void execute(EList<String> events) {
5      // Preprocess the events
6      val preprocessedEvents = events.map[toUpperCase]
7      // Call back the overriden implementation in FSMAspect
8      _self.super_execute(preprocessedEvents)
9    }
10 }
```

Listing 7.2: Aspect inheritance and method overriding in K3

Finally, the attributes declared in an aspect can be supplemented with additional annotations that precise their semantics in a modeling context. These annotations include `@Containment` to specify that a reference towards another class is a containment reference, or `@Opposite` to specify that a reference towards another class is the opposite of another reference in the target class. These annotations are taken into account when inferring the signature of an aspect (cf. Chapter 6). The resulting signature metamodel incorporates the additional semantics expressed through annotations. The documentation website of K3 lists all these annotations together with their uses.[5]

7.1.3 Assembling Abstract Syntax and Semantics in Melange

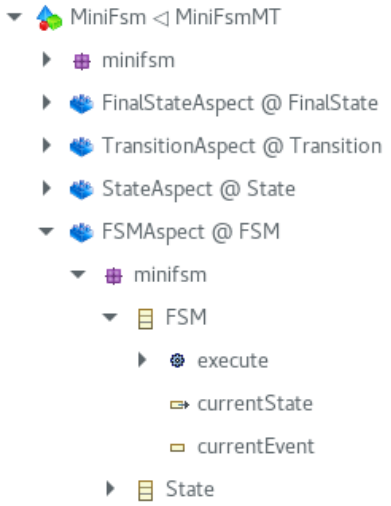
The two artifacts presented in Sections 7.1.1 and 7.1.2 are defined separately to foster separation of concerns in language development. This means that the same set of aspects can be applied to different metamodels, if they expose the appropriate concepts. Conversely, different sets of aspects can be applied to the same metamodel, which provides a mean to define different semantics for the same abstract syntax (e.g., an interpreter, a set of validation rules, and a compiler). Despite being defined separately, the abstract syntax and

[5] http://diverse-project.github.io/k3/

semantics of a language must ultimately be assembled together to form the overall language's implementation. As illustrated in Figure 7.2a, new languages are declared using the **language** keyword. Each **language** has a unique name and resides in a given **package**. Within a **language**, language designers can import existing Ecore files using the **syntax** keyword and weave aspects using the **with** keyword. If multiple **syntax** are imported, they are merged together. Different validation rules ensure that the imported artifacts are compatible one another, e.g., that there is no conflicting concept in two syntax specifications, or that each aspect has a corresponding meta-class in the language's syntax (as explained in Chapter 6). If any validation rule is violated, Melange reports an error to the user detailing the problem. The `MiniFsm` language is defined as depicted in Figure 7.2a, where each imported aspect defines the operational semantics of the corresponding concepts `FSM`, `State`, and `Transition` defined in `"MiniFsm.ecore"`.

```
1  package minifsm
2
3  language MiniFsm {
4    syntax "MiniFsm.ecore"
5    with semantics.FinalStateAspect
6    with semantics.TransitionAspect
7    with semantics.StateAspect
8    with semantics.FSMAspect
9  }
```

(a) Melange specification of `MiniFsm`



(b) Corresponding outline view

Figure 7.2: Assembling the `MiniFsm` language in Melange

On the user interface side, the Melange editor comes with a dedicated outline that highlights the different constituents of a language, as depicted in Figure 7.2b. Here, the `MiniFsm` language is composed of its abstract syntax represented as a package `minifsm`, and a set of aspects, each woven (denoted `@`) on a meta-class of its abstract syntax. Each aspect contains another package corresponding to its signature in the form of a metamodel, as explained in Chapter 6.

7.1.4 Language Extension and Composition

Melange distinguishes itself from over language workbenches by considering languages as first-class entities that can be extended and composed. In this section, we build on the `MiniFsm` language to illustrate typical language extension and composition scenarios. The reader can refer to Chapter 6 for a precise definition of the semantics of the composition operators of Melange.

Language extension In Melange, language extension is realized with the `inherits` keyword. Similarly to class inheritance in object-oriented programming, language inheritance allows to reuse the definition of a previous language as the basis for a new one. Additionally, Melange ensures that the sub-language remains compatible with the tools of the super-language, i.e., that its exact model type is a subtype of the exact model type of its super-language (cf. Chapter 5).

This mechanism is often used to extend a language with its execution semantics, as depicted in Listing 7.3. Here, we create an executable variant of the `MiniFsm`. Because `ExecutableMiniFsm` inherits from `MiniFsm`, the tools and services defined on `MiniFsm` can be reused on `ExecutableMiniFsm`. For instance, editors defined on `MiniFsm` are immediately available on `ExecutableMiniFsm`, and new tools such as an animator or a simulator can be defined on `ExecutableMiniFsm`.

```
1 language MiniFsm {
2   syntax "MiniFsm.ecore"
3 }
4 language ExecutableMiniFsm inherits MiniFsm {
5   with semantics.FinalStateAspect
6   with semantics.TransitionAspect
7   with semantics.StateAspect
8   with semantics.FSMAspect
9 }
```

Listing 7.3: Defining the executable variant of `MiniFsm`

Alternatively, Listing 7.4 depicts another inheritance scenario where the `MiniFsm` language is extended to override one of the computation step defining its semantics in a new `MiniFsmVariant` language. `MiniFsmVariant` reuses all the syntax and semantics of `MiniFsm`, and weaves the new aspect `FSMAspectOverride` depicted in Listing 7.2 to override the `execute` method woven on the `FSM` metaclass. The `syntax` keyword can also be used in sub-languages to augment previous syntax definitions. In this case, the metamodel imported in the sub-language is merged with the metamodel defining the syntax of its super-language. Multiple inheritance is also supported, in which case the different languages are linearized following the rules described in Chapter 6.

```
1 language MiniFsm { /* cf. Listing 7.2a */ }
2 language MiniFsmVariant inherits MiniFsm {
3   with semantics.FSMAspectOverride
4 }
```

Listing 7.4: Extending the `MiniFsm` language to define a semantic variation point

Language composition To present the composition mechanisms of Melange, we introduce the new language `MiniActionLang`. `MiniActionLang` is a minimalistic action language that supports the expression of blocks of code, consisting of a set of statements and expressions. It supports basic definition and manipulation of Boolean and integer variables, and basic control flow constructs such as conditional `if` statements and `while` loops. In Melange, `MiniActionLang` is defined as shown in Listing 7.5. The `"MiniActionLang.ecore"` metamodel describes concepts such as `IntegerVariable`, `If`, `Block`, etc. The `"RuntimeConcepts.ecore"` metamodel describes the runtime concepts of the action language, such as the current value of variables. The merge of these two Ecore files (Lines 2–3), along with the aspects that define their semantics (Line 4) form the overall language implementation.[6]

```
1 language MiniActionLang {
2   syntax "MiniActionLang.ecore"
3   syntax "Context.ecore"
4   with minilang.semantics.*
5 }
```

Listing 7.5: The `MiniActionLang` language in Melange

`MiniFsm` lacks the expressiveness required to express guards on transitions, and actions in states. The composition operators of Melange can be used to solve these problems by composing the `MiniActionLang` language with the `MiniFsm` language in a meaningful way to create a new language `Fsm`, as depicted in Listing 7.6.

[6]The wildcard character `*` in Line 4 specifies that all the aspects defined in the `minilang.semantics` package must be imported.

```
1 language Fsm inherits MiniFsm {
2   slice MiniActionLang on ["BooleanExpression", /* ... */]
3     renaming { "miniactionlang" to "minifsm" }
4   with glue.StateGlue
5   with glue.TransitionGlue
6 }
```

Listing 7.6: Composing `MiniFsm` and `MiniActionLang` to form the `Fsm` language

Here, we create a new variant of the `MiniFsm` language (Line 1). This new variant is complemented with a subset of the `MiniActionLang` language, where only the relevant concepts are kept. Because the control flow of the new language is defined by the states and transitions of the state machine, the `If` and `While` statements of `MiniActionLang` are pruned using the **`slice`** keyword (Lines 2). The **`slice`** clause extracts, from a given language, the syntax matching the slicing criterion specified after the **`on`** clause, and the associated aspects. The **`renaming`** clause then ensures that all the concepts gathered from both `MiniFsm` and `MiniActionLang` end up in the same package `"minifsm"`. Finally, two new aspects `StateGlue` and `TransitionGlue` are woven on the resulting language to glue together the two languages at the syntax and semantic levels. The code of these two aspects is given in Listing 7.7 with appropriate comments.

```
1  @Aspect(className = State)
2  class StateGlue {
3    // Each State contains a Block of code (from MiniActionLang)
4    @Containment Block block
5
6    // Executing a state boils down to executing its Block
7    @OverrideAspectMethod
8    override void execute() {
9      _self.block.execute()
10   }
11 }
12
13 @Aspect(className = Transition)
14 class TransitionGlue {
15   // Each Transition can have a BooleanExpression guard
16   // (from MiniActionLang)
17   @Containment BooleanExpression guard
18
19   // A Transition can be fired if the guard evaluates to true
20   @OverrideAspectMethod
21   override boolean isFireable() {
22     return _self.super_isFireable() &&
23            (_self.guard === null  _self.guard.eval())
24   }
25 }
```

Listing 7.7: The `StateGlue` and `TransitionGlue` aspects

7.2 Model Types

As seen in Chapters 5 and 6, every language in Melange implements its exact model type, a structural language interface exposing all its features. Melange automatically infers the exact model type of each language and, by convention, names it with the name of the language suffixed by `MT`. From Figure 7.2a, Melange thus generates a new model type named `MiniFsmMT`. The name and URI of the inferred model type can be overridden using the clause **`exactType`** `<mt-name>` `[`**`uri`** `<mt-url>]` in a language definition. From the language specification depicted in Figure 7.2a, Melange generates the model type depicted in Figure 7.3. The new features and methods woven by the aspects defining the semantics of `MiniFsm` are highlighted in gray.

Model types can also be defined explicitly using the **`modeltype`** keyword. In this case, the Ecore file imported using the **`syntax`** keyword defines the concepts and features of the model type. This feature is typically used to follow a design-by-contract method for language definition. In this situation, a language designer first explicitly defines one or several model types, and then

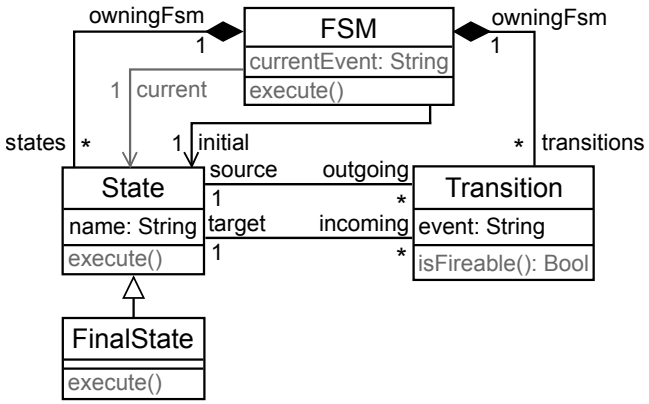
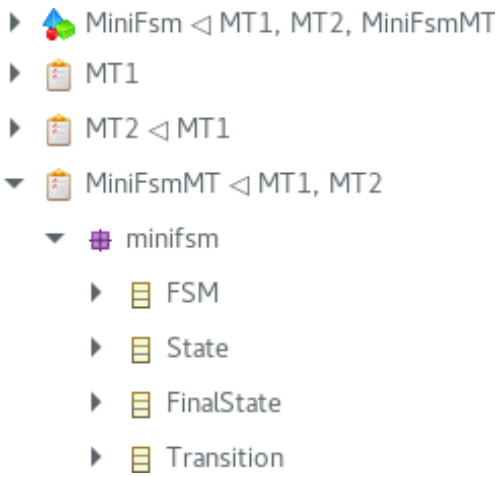


Figure 7.3: `MiniFsm`'s exact model type: `MiniFsmMT`

implement a new language definition that implements these model types. Doing so, she ensures that her language implements the appropriate interfaces, and can thus benefit from the tools and services defined on these interfaces. This situation is depicted in Figure 7.4a. If the type checker of Melange detects that one of the model type pointed by the **`implements`** keyword is not implemented by the language, an error listing the elements that do not match is reported to the user.

```
1  package minifsm
2
3  modeltype MT1 {
4    syntax "MT1.ecore"
5  }
6
7  modeltype MT2 {
8    syntax "MT2.ecore"
9  }
10
11 language MiniFsm implements
12   MT1, MT2 {
13   syntax "MiniFsm.ecore"
14 }
```

(a) Explicit model type definition and implementation in Melange



(b) Corresponding outline view

Figure 7.4: Model types in Melange

On the user interface side, the outline shows the inferred and hand-crafted model types, as well as the implementation relations between languages and model types. From the Melange specification of Figure 7.2a, the outline is as

depicted in Figure 7.4b. Each model type contains a package that describes the concepts and features it exposes. The ◁ symbol denotes an implementation relation between a language and a model type, or a subtyping relation between two model types. Here, `MT2` is a subtype of `MT1`, and `MiniFsmMT` a subtype of `MT2`. `MiniFsm` implements all of them: the implementation relation towards `MT1` and `MT2` is explicitly requested in Figure 7.4a, while the implementation relation towards `MiniFsmMT` is automatically inferred by Melange.

7.3 Integration with Eclipse and EMF

EMF is a rich ecosystem that includes numerous modeling technologies such as transformation engines, graphical and textual editors, and persistence frameworks. Throughout the development of Melange, we have paid a particular attention to the seamless integration of Melange with the EMF ecosystem. This integration is beneficial for both parts. First, Melange relies internally on EMF technologies for the definition of languages (such as Ecore), and can thus benefit from a rich diversity of existing metamodels. Second, any language designer familiar with the EMF ecosystem can benefit from the composition and interoperability facilities provided by Melange, without disrupting her workflow.

As detailed in the remainder of this section, the integration of Melange and EMF relies on two main pillars. First, any language generated by Melange (e.g., the result of the extension of an existing language) is itself a plain EMF language that can be manipulated using other EMF technologies (Section 7.3.1). Second, the *MelangeResource*, as introduced in Chapter 5, can be used to provide model polymorphism capabilities to any EMF-based technology transparently (Section 7.3.2).

7.3.1 Compilation Scheme

In Eclipse, a contextual menu on Melange files enables users to request the generation of the runtime of each language defined in a Melange file. For each language, the compiler of Melange generates a new Eclipse plug-in project containing (i) an Ecore file describing its abstract syntax and another describing its exact model type in the `model` directory (ii) the Java implementation generated by EMF from the Ecore files in the `src` directory (iii) the aspects implementing its semantics in the `src-gen` directory and (iv) a `plugin.xml` file that contributes several extension points used to register the language in Eclipse. The typical structure of a language project generated by Melange is depicted in Figure 7.5.

When deployed as a plug-in in a new Eclipse instance, each project automatically registers the corresponding languages and model types. Each language is registered as a classic EMF project. Users can thus leverage other tools of the EMF ecosystem, such as a tree editor or a graphical editor, to edit

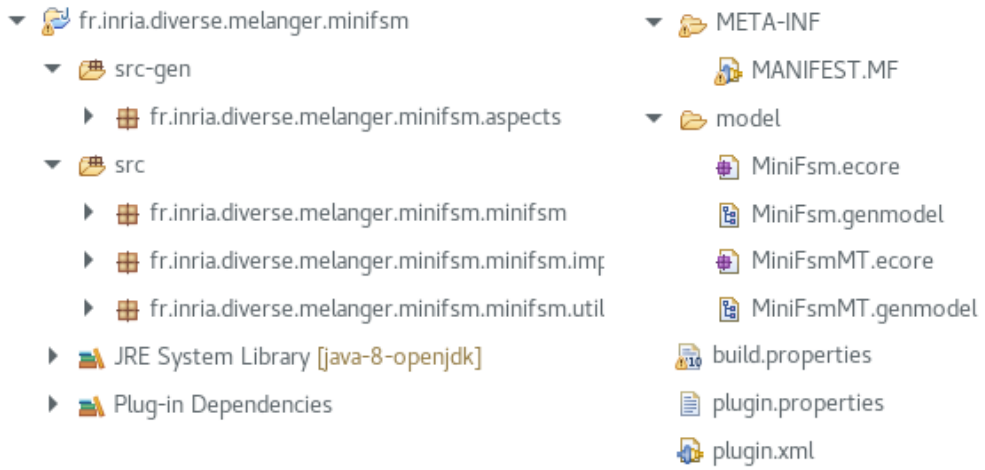


Figure 7.5: Eclipse project generated from the `MiniFsm` language of Figure 7.2a

models conforming to it. They can also define new model transformations, either directly on the metamodels, or on the model types. In the latter case, combined with the *MelangeResource*, these transformations are inherently generic and can be reused for any other registered language that implements the appropriate model type.

7.3.2 The *MelangeResource*

The *MelangeResource*, presented in Chapter 5, is a mechanism integrated with EMF that transparently provides model polymorphism to any EMF-based tool of the Eclipse modeling ecosystem. In this section, we present how the *MelangeResource* is used to polymorphically invoke tools and services defined around a DSL, for instance implemented as Java, ATL [150], or QVTo [211] transformations.

Illustration with Java Listing 7.8 depicts how to invoke Java transformations polymorphically using the *MelangeResource*. The methods `execute` and `prettyPrint` are defined as taking as parameter an object typed by the object type `FSM` of `MiniFsmMT` (cf. Figure 7.3). The method `execute` simply delegates the execution of the state machine to the operational semantics using the `execute` method woven on `FSM`. Dynamic dispatch applies here, and the most precise implementation of the `execute` method is used. For instance, if this transformation is applied to a model conforming to the `MiniFsmVariant` language, the `execute` method of the `FSMAspectOverride` aspect is invoked (cf. Listing 7.4). The `reflexivePrettyPrint` method illustrates that model polymorphism also works when using the reflexive API of EMF [240]. In this case, the reflexive API calls (e.g., `eClass()`) are catched directly at the model type level: the information returned describes the `MiniFsmMT` model type (cf.

Figure 7.3). When the developer uses this information to manipulate the model (e.g., through the `eGet` method, Line 27), the calls are dispatched directly to the language the model conforms to (in this case, either `MiniFsm` or `Fsm`).

The `main` method loads two different models: `Simple.minifsm` conforms to the `MiniFsm` language (cf. Figure 7.2a), and `Simple.fsm` conforms to the `Fsm` language (cf. Listing 7.6). Both implement the `MiniFsmMT` model type and can invoke the three tranformation methods.

```
1  package minifsmtest;
2
3  import minifsm.fsmmt.FSM;
4
5  public class PolymorphicFsm {
6    // execute() and prettyPrint() specify as parameter
7    // the FSM object type of MiniFsmMT
8    public static void execute(FSM fsm) {
9      System.out.print("Output:␣");
10     // Delegating execution to the corresponding aspect
11     fsm.execute("adcdce");
12     System.out.println();
13   }
14   public static void prettyPrint(FSM fsm) {
15     fsm.getStates().forEach(s -> {
16       System.out.println("State" + s.getName());
17       s.getIncoming().forEach(t -> {
18         System.out.println("\tIncoming:␣" + t.getEvent());
19       });
20       s.getOutgoing().forEach(t -> {
21         System.out.println("\tOutgoing:␣" + t.getEvent());
22       });
23     });
24   }
25   public static void reflexivePrettyPrint(EObject o) {
26     o.eClass().getEStructuralFeatures().forEach(f -> {
27       System.out.println(f.getName() + "␣=␣" + o.eGet(f));
28       o.eContents().forEach(c -> reflexivePrettyPrint(c));
29     });
30   }
31   public static void main(String[] args) {
32     ResourceSet rs = new ResourceSetImpl();
33     // Load a MiniFsm model and return it as typed by MiniFsmMT
34     Resource mfsmRes  = rs.getResource(URI.createURI(
35       "melange:/file/input/Simple.minifsm?mt=minifsm.MiniFsmMT"), true);
36     // Load a Fsm model and return it as typed by MiniFsmMT
37     Resource fsmRes  = rs.getResource(URI.createURI(
38       "melange:/file/input/Simple.fsm?mt=minifsm.MiniFsmMT"), true);
39     // The root of both models is thus typed by the FSM
40     // object type of MiniFsmMT
41     FSM mfsmRoot = (FSM) mfsmRes.getContents().get(0);
42     FSM fsmRoot  = (FSM)  fsmRes.getContents().get(0);
43     // Polymorphic execution
44     System.out.println("execute:\n");
45     execute(mfsmRoot);
46     execute(fsmRoot);
47     // Polymorphic pretty-print
48     System.out.println("pretty-print:\n");
49     prettyPrint(mfsmRoot);
50     prettyPrint(fsmRoot);
51     // Polymorphic reflexive pretty-print
```

```
52      System.out.println("reflexivePrettyPrint:\n");
53      reflexivePrettyPrint(mfsmRoot);
54      reflexivePrettyPrint(fsmRoot);
55    }
56  }
```

Listing 7.8: Implementing and invoking polymorphic model transformations in Java

Illustration with ATL Every EMF-based tool of the Eclipse modeling ecosystem loads models using the EMF API, in a similar way than depicted in Listing 7.8. Because metamodels and model types are described by Ecore files, defining an ATL transformation on a model type is done in the exact same way than on a metamodel. Listing 7.9 depicts a simple ATL transformation that takes as input a model typed by the `MiniFsmMT` model type, and produces as output a new `MiniFsm` model where the transitions are inverted. This transformation can be applied to any model conforming to a language that implements the `MiniFsmMT` model type (e.g., `MiniFsm`, `Fsm`, or `MiniFsmVariant`). To do so, the user must invoke the ATL transformation with the proper Melange URI, which can be specified either directly in the graphical *launch configuration* of an ATL transformation, or when invoking the ATL transformation programmatically.

```
1  -- @nsURI MiniFsm=http://minifsm/
2  -- @nsURI MiniFsmMT=http://minifsmmt/
3
4  module DummyInvert;
5  create OUT : MiniFsm from IN : MiniFsmMT;
6
7  rule InvertFsm {
8    from inputFsm  : MiniFsmMT!FSM
9    to   outputFsm : MiniFsm!FSM (
10     states  <- inputFsm.state,
11     initial <- inputFsm.finalState->first(),
12     final   <- inputFsm.initial
13   )
14 }
15 rule InvertStates {
16   from inputState  : MiniFsmMT!State
17   to   outputState : MiniFsm!State (
18     name <- 'Inverted' + inputState.name
19   )
20 }
21 rule InvertTransitions {
22   from inputTrans  : MiniFsmMT!Transition
23   to   outputTrans : MiniFsm!Transition (
24     event  <- 'Inverted' + inputTrans.event,
25     source <- inputTrans.target,
26     target <- inputTrans.source
27   )
28 }
```

Listing 7.9: A generic model the ransformation in ATL

Illustration with QVTo Just like ATL, QVTo transformations can be defined on a model type.[7] Listing 7.10 depicts a QVTo transformation equivalent to the ATL transformation of Listing 7.9. Here again, the transformation can be applied to any model conforming to a language that implements the `MiniFsmMT` model type, using a Melange URI. This URI can be specified through QVTo's launch configuration interface, or programmatically.

[7]Note that QVTo's **`modeltype`** keyword should not be confused with Melange's **`modeltype`** keyword.

```
1  modeltype MiniFsmMT uses 'http://minifsmmt/';
2  modeltype MiniFsm   uses 'http://minifsm/';
3
4  transformation dummyInvert(
5    in inFsm : MiniFsmMT, out outFsm : MiniFsm);
6
7  main() {
8    inFsm.rootObjects()[MiniFsmMT::FSM] -> map mapFSM();
9  }
10 mapping FsmMT::FSM::mapFSM() : Fsm::FSM {
11   ownedState   := self.states -> map mapState();
12   initialState := self.finalState -> first().map mapState();
13   finalState   := self.initial.map mapState();
14 }
15 mapping MiniFsmMT::State::mapState() : MiniFsm::State {
16   name := "Inverted" + self.name;
17   outgoing := self.incoming -> map mapTransition();
18 }
19 mapping MiniFsmMT::Transition::mapTransition()
20   : MiniFsm::Transition {
21   event  := "Inverted" + self.event
22   target := self.source.map mapState();
23   source := self.target.map mapState();
24 }
```

Listing 7.10: A generic model transformation in QVTo

7.4 Conclusion

In this chapter, we have presented the Melange language workbench from a user point of view. This chapter thus complements the presentation of Melange in Chapters 5 and 6 where the internal implementation of the support for model polymorphism and modular language development is detailed. Thorough this chapter, we used the illustrative language `MiniFsm` to present syntax and semantics definition, syntax and semantics assembling, and language extension and composition. We also presented how models can be flexibly loaded by users of the language using the *MelangeResource*.

To the best of our knowledge, despite the popularity of EMF in both academia and industry, Melange is the first language workbench to provide such support for advanced language assembly, composition and interoperability in the EMF ecosystem. Melange is a key component of the GEMOC studio,[8] where it serves as an assembly and extension language for the definition of heterogeneous modeling languages. Melange is also used in the context of the

[8] http://gemoc.org/studio/

LEOC Clarity project[9] to define lightweight viewpoints on the Capella systems engineering language.[10]

[9] http://www.clarity-se.org/
[10] https://polarsys.org/capella/

Part IV

Conclusion and Perspectives

Chapter 8

Conclusion and Perspectives

In this chapter, I resume the contributions of this thesis and offer concluding remarks (Section 8.1). Then, I develop two main perspectives that directly stem from the contributions presented in this thesis (Section 8.2).

8.1 Conclusion

DSLs are software languages specifically tailored to a particular domain of application. They allow reasoning about and implementing software systems at the level of abstraction of the problem domain. The dedicated tools accompanying a DSL automatically bridge the gap between abstractions at the problem level and their implementation in specific technologies at the solution level. Both model-driven engineering and language-oriented programming rely on the definition of appropriate DSLs to address each concern in systems development. Specifically, external DSLs offer essential added value for the end users as their syntax and semantics can be fully tailored to a specific domain of application, without restriction.

In this thesis, we identified two main challenges to be addressed for the engineering of external DSLs. First, the proliferation of independently-developed and constantly-evolving DSLs raises the problem of interoperability between different DSLs and their ecosystem (IDEs, transformations, etc.). Second, since DSLs and their environment are costly to develop, the benefits in terms of productivity when using DSL technologies must offset the initial investment required in developing them. Techniques must thus be developed to ease the engineering of new DSLs.

To tackle these challenges, we first introduced the notion of language interface as a fundamental mean to raise the level of abstraction in different activities of DSLs engineering. A language interface is a relevant abstraction for a specific purpose of a language or language component. A language interface can be defined manually, or inferred automatically from a language implementation. Language implementations are linked to language interfaces through an implementation relation, and multiple languages may match the

same interface. This enables language designers to (i) reason about languages through dedicated abstractions and (ii) define generic tools and services on a language interface that can be reused for all matching languages.

Then, we used model types (a specific structural language interface) to provide a safe mechanism of model polymorphism to increase flexibility and interoperability in DSLs engineering (**Challenge #1**). We based our contribution on the theory of model typing, which is inspired by seminal works on type groups and family polymorphism in the programming community. We have shown how model polymorphism can be achieved in practice (i) to define generic transformations on a family of eight finite-state machine language variants and (ii) to support flexible manipulation of UML models gathered from Github using four different versions of the UML standard. The model polymorphism mechanism we propose is flexible yet safe, and increases interoperability between the environments of different DSLs.

Finally, to foster reuse and composition in DSLs engineering, we proposed a meta-language dedicated to modular and reusable development of DSLs from legacy language artifacts (**Challenge #2**). This meta-language provides a set of operators dedicated to DSL assembly and customization inspired from previous taxonomies of language composition in the SLE community. Furthermore, it includes a new restriction operator that is missing from this taxonomies as a way to restrict the scope and expressiveness of existing languages. We have shown how the dedicated set of operators we propose allows to engineer a new language for IoT systems modeling as an assembly of three publicly-available language implementations: the OMG Interface Description Language, the UML activity diagram, and Lua.

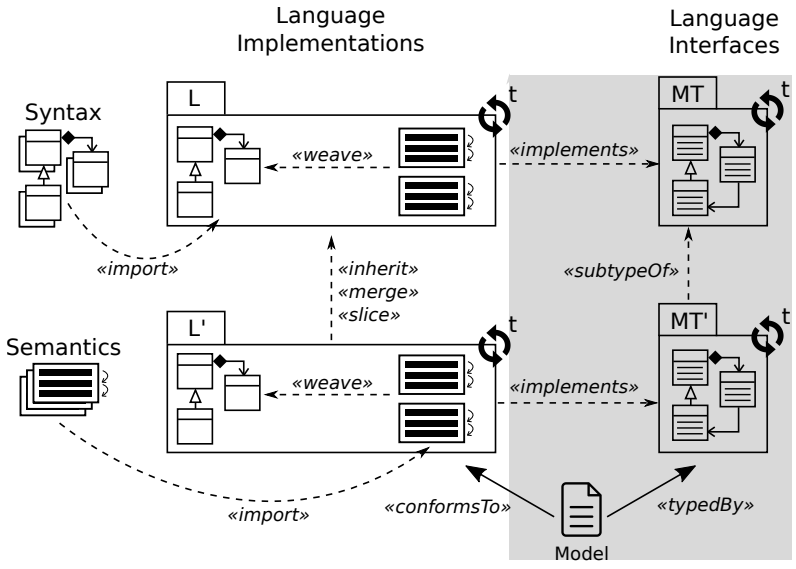


Figure 8.1: Overview of the contributions, as presented in Chapter 1

Our three contributions are highly interrelated, as depicted in Figure 8.1. Language interfaces – and more precisely model types – are used both for model polymorphism, and DSL reuse and composition. Furthermore, model polymorphism is directly used when composing DSLs to reason about their structural compatibility, the substitutability of their models, and to define generic transformations over language families created using composition operators. We implemented all our contributions in a new language workbench named Melange. Every language in Melange is associated to its structural interface materialized by a model type. The structural type system of Melange automatically infers the subtyping relation between different model types. When subtyping relations are inferred, Melange automatically provides model polymorphism and substitutability among the appropriate DSLs. Melange is seamlessly integrated with the Eclipse Modeling Framework, the *de facto* technological standard in both academia and industry. As such, Melange transparently provides model polymorphism and language composition capabilities to language designers working in this technological space. To the best of our knowledge, Melange is the first language workbench to provide such DSL engineering capabilities in the EMF ecosystem.

In conclusion, the contributions presented in this thesis address the two challenges we identified. The Melange language workbench constitutes a solid technical contribution that has been applied in different collaborative projects for different purposes: the definition of executable DSLs, the definition of generic tools and transformations, and the composition of independenty-developed language artifacts.

8.2 Perspectives

We consider this thesis as a first step towards a better understanding of composition and interoperability in DSLs engineering. Along the way, we identified many promising perspectives of our work. In this section, we choose to detail two of these long-term perspectives: component-based language engineering, and viewpoint engineering.

8.2.1 Component-Based Language Engineering

In this thesis, we specifically focused on the reuse of legacy language artifacts, including syntax, semantics, and their associated tools and services. In this context, we cannot make any assumption on the way legacy artifacts are defined (e.g., with modularity and reuse in mind). The reuse and composition capabilities we propose are thus by definition limited compared to a development method that would *anticipate* modularity and reuse.

One way to go further in this direction would be to support the definition of explicit language modules, i.e., fragments of language that cannot be used as is, but are meant to be composed by language designers to create complete

languages. This would require to be able to express explicit provided and required interfaces of language modules. As an example, a simple "for-loop" construct consists of a condition (its stop condition) and the block of code on which it iterates. So, a "for-loop" module would require a language module to express conditions (and an evaluation function), and another module to express its inner block of code (and an evaluation function). In turn, the "for-loop" module would provide the concept of for-loop and the associated evaluator. The modules, as well as the interfaces, crosscut the different concerns of a language (syntax, semantics), and can be independently developed and validated. Instead of starting from scratch, a language designer could pick some modules and compose them together *à la carte* in a meaningful way to produce a new DSL. The derivation of a new language could be for instance supported by a feature model, acting as the front-end for the configuration and derivation of a new language.

The general objective is to reduce further the development costs of DSLs by enabling DSL designers to reuse as much as possible previous (fragments of) language definitions. These fragments could be independently implemented, tested, and validated. They could possibly be written by other language designers, and available "off-the-shelf" in a language workbench. This would also enable domain experts who lack strong knowledge in language theory to build their own language easily.

Going towards component-based language engineering raises many scientific questions: what is the appropriate level of granularity for modularizing languages; what is the appropriate formalism to write the provided and required interfaces; when composing at the specification level, how to compose at the implementation level without having to re-generate everything (e.g., making the interpreters cooperate); if some properties have been formally checked on a given module, do they still hold on the composed languages; how to unify the basic data types, type systems, etc. when composing; how to handle crosscutting concerns that are inherently hard to modularize (e.g., exceptions); and many more.

We believe some of the contributions presented in this thesis could be leveraged to realize this vision. The notion of language interface may be used to realize the concept of provided and required interfaces of a language: model types, for instance, allow to abstract away some of the intrinsic complexity of language implementations, and to reason about language modules substitutability and composition. The composition and customization operators may then be used to realize the concrete composition of different modules, while possibly customizing them to fit a new particular domain of application.

8.2.2 Viewpoint Engineering

In software and systems engineering, the idea of viewpoints is to materialize multiple perspectives in the development of complex systems. Each viewpoint

typically materializes a stakeholder's point of view on a system, tailored to her specific needs. While the idea of viewpoint engineering is not new, it garnered a considerable interest in the last years through the definition of the ISO/IEC 42010. The ISO/IEC 42010 aims to standardize the description of architectures in systems engineering and heavily relies on the notion of viewpoints. In the following paragraphs, we detail our vision on the interaction between software language engineering and viewpoint engineering.

In model-driven engineering, separation of concerns is supported by the use of dedicated DSLs that materialize the natural decomposition of the system in different domains of expertise. For instance, a language dedicated to variability modeling typically includes the concepts of variant, feature, configuration, etc. The interaction between the user of a language and the language itself is supported by the use of tools that the user leverages to undertake concrete *tasks* (e.g., editing a model, analyzing it, generating artifacts, etc.). In the variability modeling example, different stakeholders have different interests and different ways to manipulate the same underlying concepts. Some want to manipulate the modeled features through a tree structure editor, others as product-comparison matrices, others using a configurator for deriving specialized products.

More generally, we observe that the separation of concerns in different DSLs is not always natural for the realization of concrete tasks. The use of languages indeed varies according to the particular task at hand. Since each stakeholder interacts with a given language in a particular context, the use of a language must vary according to its users. Viewpoint engineering aims at adjusting the interaction with languages to the need of specific stakeholders. From the same underlying concepts (e.g., the ones specific to variability modeling), different viewpoints provide the most appropriate views of the same system to carry on specific tasks. Thereby, viewpoint engineering supports a new layer of decomposition, namely the *separation of tasks*, atop the separation of concerns. Figure 8.2 depicts the interaction between software language engineering and viewpoint engineering.

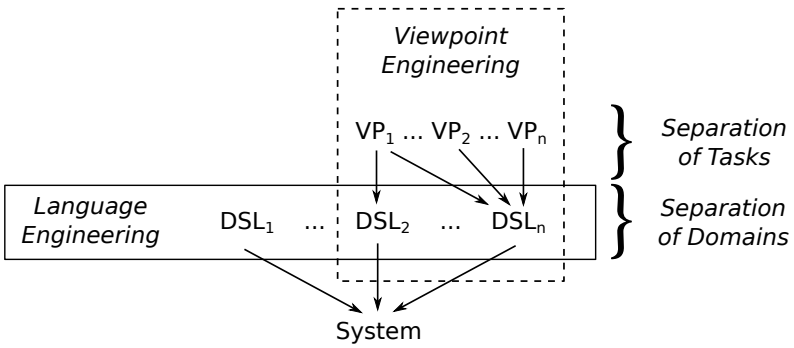


Figure 8.2: Software language engineering and viewpoint engineering

Naturally, software language engineering and viewpoint engineering are highly complementary. The former provides the appropriate abstractions leveraged by the latter to provide adapted views for manipulating the underlying models. While many viewpoints may be defined on the same language, some others are orthogonal to the decomposition of domains materialized in different languages. Some tasks, such as security and safety analysis or global simulation indeed crosscut multiple domains of expertise. For instance, in the variability modeling example, a particular stakeholder may have to relate abstract features of the variability model to functional artifacts expressed for instance in SysML.

The definition of a new viewpoint may involve the projection (i.e., selection) of information from base DSLs or the insertion of new syntactic and semantic elements only relevant for a particular task. The operators for language extension, restriction, and customization presented in this thesis are thus particularly relevant in this context as they provide a way to manipulate languages as first-class entities. In complement, the idea of language interfaces, and in particular model types, may be useful for the definition of viewpoints. Model types allow to filter information from a language, by exposing only a subset of the features of its abstract syntax and semantics. Moreover, transformations can be directly defined upon a model type to define the tools and services supporting particular tasks of a given stakeholder. The language interface materialize a stakeholder's view on the system, and the tools and services define how she interact with the underlying models.

The definition of viewpoints on a particular DSL is already a reality, e.g., in systems engineering. Capella, the systems engineering language developed by Thales Group and part of the PolarSys initiative[1], for instance, is already complemented with a framework for viewpoint engineering named KitAlpha[2]. In the context of the LEOC Clarity project, first experiments show that Melange could be used as an alternative to KitAlpha for the design and manipulation of viewpoints on the Capella language.

[1] https://www.polarsys.org/
[2] https://polarsys.org/kitalpha/

List of Figures

List of Tables

List of Listings

Bibliography

[1] M. Acher, B. Combemale, and P. Collet. Metamorphic domain-specific languages: A journey into the shapes of a language. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 243–253. ACM, 2014.

[2] M. Alanen, I. Porres, et al. *A relation between context-free grammars and meta object facility metamodels*. Turku Centre for Computer Science, 2004.

[3] N. Amálio, J. de Lara, and E. Guerra. Fragmenta: A theory of fragmentation for mde. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 106–115. IEEE, 2015.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business process execution language for web services, 2003.

[5] S. Apel, D. Batory, and M. Rosenmüller. On the structure of crosscutting concerns: Using aspects or collaborations. In *GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2006.

[6] V. Aranega, J. Mottu, A. Etien, T. Degueule, B. Baudry, and J. Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing, Verification and Reliability*, 25(5-7): 653–683, 2015. doi: 10.1002/stvr.1532.

[7] C. Atkinson and T. Kühne. Profiles in a Strict Metamodeling Framework. *Science of Computer Programming (SCP)*, 44(1):5–22, 2002.

[8] C. Atkinson and T. Kühne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[9] C. Atkinson and T. Kühne. A tour of language customization concepts. *Advances in Computers*, 70:105–161, 2007.

[10] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957.

[11] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, et al. Report on the algorithmic language algol 60. *Numerische Mathematik*, 2(1):106–136, 1960.

[12] P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context—motorola case study. In *International Conference on Model Driven Engineering Languages and Systems*, pages 476–491. Springer, 2005.

[13] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[14] E. Barrett, C. F. Bolz, and L. Tratt. Approaches to interpreter composition. *Computer Languages, Systems & Structures*, 44:199–217, 2015.

[15] E. Barrett, C. F. Bolz, L. Diekmann, and L. Tratt. Fine-grained language composition: A case study. In *Proceddings of the 30th European Conference on Object-Oriented Programming (ECOOP'16)*, 2016.

[16] A. Benelellam, M. Tisi, J. Sanchéz Cuadrado, J. de Lara, and J. Cabot. Efficient Model Partitioning for Distributed Model Transformations. In *Proceedings of the 9th International Conference on Software Language Engineering (SLE'16)*, 2016.

[17] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing: A controlled experiment. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2016.

[18] A. Bergmayr and M. Wimmer. Generating metamodels from grammars by chaining translational and by-example techniques. In *MDEBE@ MoDELS*, pages 22–31, 2013.

[19] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform resource identifiers (URI): Generic Syntax. https://www.ietf.org/rfc/rfc2396.txt, 1998.

[20] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[21] J. Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[22] J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[23] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 273–280, 2001.

[24] J. Bézivin, F. Jouault, and D. Touzet. Principles, Standards and Tools for Model Engineering. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 28–29, 2005.

[25] A. Biboudis, P. Inostroza, and T. van der Storm. Recaf: Java dialects as libraries. In *Proceedings of the 15th International Conference on Generative Programming and Component Engineering (GPCE'16)*, 2016.

[26] A. Blouin and O. Beaudoux. Improving Modularity and Usability of Interactive Systems with Malai. In *Proceedings of the 2nd Symposium on Engineering interactive computing systems (EICS'10)*, pages 115–124, 2010.

[27] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling model slicers. In *International Conference on Model Driven Engineering Languages and Systems*, pages 62–76. Springer, 2011.

[28] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Kompren: Modeling and Generating Model Slicers. *Software and Systems Modeling (SoSyM)*, pages 1–17, 2012.

[29] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. *Centaur: the system.* ACM, 1988.

[30] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry. Supporting efficient and advanced omniscient debugging for xdsmls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–148. ACM, 2015.

[31] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, Amsterdam*, 2016. Tool demonstration.

[32] C. Brabrand and M. I. Schwartzbach. *Growing languages with metamorphic syntax macros*, volume 37. ACM, 2002.

[33] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of computer programming*, 72(1):52–70, 2008.

[34] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–157. ACM, 2016.

[35] K. B. Bruce and J. C. Vanderwaart. Semantics-driven language design:: Statically type-safe virtual types in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 20:50–75, 1999.

[36] H. Bruneliere, J. Garcia, P. Desfray, D. E. Khelladi, R. Hebig, R. Bendraou, and J. Cabot. On lightweight metamodel extension to support modeling tools agility. In *Modelling Foundations and Applications*, pages 62–74. Springer, 2015.

[37] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical Evidence about the UML: a Systematic Literature Review. *Software: Practice and Experience (SPE)*, 41(4):363–392, 2011.

[38] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proceedings of the 4th Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 457–467. ACM, 1989.

[39] L. Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79. ACM, 1988.

[40] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[41] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In *International Symposium on Theoretical Aspects of Computer Software*, pages 244–272. Springer, 1994.

[42] W. Cazzola and D. M. Olivares. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing*, 2016.

[43] W. Cazzola and E. Vacchi. Language components for modular dsls using traits. *Computer Languages, Systems & Structures*, 45:16–34, 2016.

[44] F. Chauvel and J.-M. Jézéquel. Code generation from UML models with semantic variation points. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05)*, pages 54–68. Springer, 2005.

[45] B. Cheng, B. Combemale, R. France, J.-M. Jézéquel, and B. Rumpe. *Globalizing Domain-Specific Languages*. Springer, 2015.

[46] B. H. Cheng, B. Combemale, R. B. France, J.-M. Jézéquel, and B. Rumpe. On the globalization of domain-specific languages. In *Globalizing Domain-Specific Languages*, pages 1–6. Springer, 2015.

[47] B. H. C. Cheng, T. Degueule, C. Atkinson, S. Clarke, U. Frank, P. J. Mosterman, and J. Sztipanovits. Motivating use cases for the globalization of dsls. In *Globalizing Domain-Specific Languages*, pages 21–42. Springer, 2014. doi: 10.1007/978-3-319-26172-0_3.

[48] M. Churchill, P. D. Mosses, and P. Torrini. Reusable components of semantic specifications. In *Proceedings of the 13th international conference on Modularity*, pages 145–156. ACM, 2014.

[49] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*, pages 222–231. IEEE, 2008.

[50] T. Clark. Xpl: A language for modular homogeneous language embedding. *Science of Computer Programming*, 98:589–616, 2015.

[51] T. Clark, A. Evans, and S. Kent. Engineering modelling languages: A precise meta-modelling approach. In *International Conference on Fundamental Approaches to Software Engineering*, pages 159–173. Springer, 2002.

[52] T. Clark, P. Sammut, and J. Willans. *Applied metamodelling: a foundation for language driven development.* Ceteva, 2008.

[53] T. Clark, M. Van den Brand, B. Combemale, and B. Rumpe. Conceptual model of the globalization for domain-specific languages. In *Globalizing Domain-Specific Languages*, pages 7–20. Springer, 2015.

[54] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[55] M. Clavreul. *Model and Metamodel Composition: Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques*. PhD thesis, Université Rennes 1, 2011.

[56] T. Cleenewerck. Component-based dsl development. In *International Conference on Generative Programming and Component Engineering*, pages 245–264. Springer, 2003.

[57] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *ACM Sigplan Notices*, volume 35, pages 130–145. ACM, 2000.

[58] B. Combemale. *Towards Language-Oriented Modeling*. Accreditation to supervise research, Université de Rennes 1, Dec. 2015. URL https://hal.inria.fr/tel-01238817.

[59] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux. Essay on semantics definition in mde-an instrumented approach for model verification. *Journal of Software*, 4(9):943–958, 2009.

[60] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, 2014.

[61] B. Combemale, J. DeAntoni, O. Barais, C. Brun, A. Blouin, T. Degueule, E. Bousse, and D. Vojtisek. A solution to the ttc'15 model execution case using the GEMOC studio. In *Proceedings of the 8th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences, L'Aquila, Italy, July 24, 2015.*, pages 19–26, 2015. URL http://ceur-ws.org/Vol-1524/paper12.pdf.

[62] V. Cosentino, M. Tisi, and J. L. C. Izquierdo. A model-driven approach to generate external dsls from object-oriented apis. In *SOFSEM 2015: Theory and Practice of Computer Science*, pages 423–435. Springer, 2015.

[63] M. L. Crane and J. Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05)*, pages 97–112. Springer, 2005.

[64] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. Viatra-visual automated transformations for formal verification and validation of uml models. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 267–270. IEEE, 2002.

[65] J. S. Cuadrado, E. Guerra, and J. de Lara. A Component Model for Model Transformations. *IEEE Transactions on Software Engineering*, 40(11):1042–1060, 2014.

[66] J. S. Cuadrado, E. Guerra, and J. de Lara. Reverse engineering of model transformations for reusability. In *International Conference on Theory and Practice of Model Transformations*, pages 186–201. Springer, 2014.

[67] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard. Templatable Metamodels for Semantic Variation Points. In *Proceedings of the 3rd European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA'07)*, pages 68–82, 2007.

[68] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[69] J. De Lara and E. Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.

[70] J. De Lara and E. Guerra. Generic Meta-modelling with Concepts, Templates and Mixin Layers. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, pages 16–30, 2010.

[71] J. De Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer, 2002.

[72] J. de Lara, E. Guerra, and J. Sanchez Cuadrado. A-posteriori typing for model-driven engineering. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 156–165. IEEE, 2015.

[73] J. De Lara, E. Guerra, and J. Sánchez Cuadrado. A-posteriori Typing for Model-Driven Engineering. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 156–165, 2015.

[74] T. Degueule. Interoperability and composition of dsls with melange. Technical report, Inria, 2016. Submission to the ACM Student Research Competition Grand Finals.

[75] T. Degueule, B. Combemale, A. Blouin, and O. Barais. Reusing legacy dsls with melange. In *Proceedings of the 15th Workshop on Domain-Specific Modeling, DSM@SPLASH 2015, Pittsburgh, PA, USA, October 27, 2015*, pages 45–46, 2015. doi: 10.1145/2846696.2846697.

[76] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J. Jézéquel. Melange: a meta-language for modular and reusable development of dsls.

In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, pages 25–36, 2015. doi: 10.1145/2814251.2814252.

[77] T. Degueule, J. B. F. Filho, O. Barais, M. Acher, J. L. Noir, S. Madelénat, G. Gailliard, G. Burlot, and O. Constant. Tooling support for variability and architectural patterns in systems engineering. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 361–364, 2015. doi: 10.1145/2791060.2791097. Tool demonstration.

[78] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J. Jézéquel. Safe model polymorphism for flexible modeling. *Computer Languages, Systems & Structures*, 2016. ISSN 1477-8424. doi: http://dx.doi.org/10.1016/j.cl.2016.09.001.

[79] T. Degueule, B. Combemale, and J.-M. Jézéquel. On language interfaces. In *PAUSE: Present and Ulterior Software Engineering*. Springer, 2017. To appear.

[80] D. Di Ruscio, L. Iovino, and A. Pierantonio. Coupled Evolution in Model-driven Engineering. *IEEE Software*, 29(6):78–84, 2012.

[81] L. Diekmann and L. Tratt. Eco: a Language Composition Editor. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE'14)*, pages 82–101. Springer, 2014.

[82] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *Software & Systems Modeling*, 7(4):443–467, 2008.

[83] I. DiverSE Team. The melange language workbench, 2016. URL `http://melange-lang.org`.

[84] L. M. do Nascimento, D. L. Viana, P. A. S. Neto, D. A. Martins, V. C. Garcia, and S. R. Meira. A systematic mapping study on domain-specific languages. In *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA'12)*, pages 179–187, 2012.

[85] Eclipse. Sirius. `http://www.eclipse.org/sirius/`.

[86] M. Egea and V. Rusu. Formal Executable Semantics for Conformance in the MDE Framework. *Innovations in Systems and Software Engineering*, 6(1-2):73–81, 2010.

[87] T. Ekman and G. Hedin. The jastadd system—modular extensible compiler construction. *Science of Computer Programming*, 69(1):14–26, 2007.

[88] M. Emerson and J. Sztipanovits. Techniques for metamodel composition. In *OOPSLA–6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[89] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: library-based syntactic language extensibility. In *Proceedings of the 26th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'11)*, pages 391–406. ACM, 2011.

[90] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the 12th Workshop on Language Descriptions, Tools, and Applications*, page 7. ACM, 2012.

[91] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[92] E. Ernst. Family Polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 303–326. Springer, 2001.

[93] M. Eysholdt and H. Behrens. Xtext: Implement your Language Faster than the Quick and Dirty Way. In *Proceedings of the International Conference on Object-Oriented Programming Systems Languages and Applications Companion (OOPSLA'10 Companion)*, pages 307–309. ACM, 2010.

[94] J.-M. Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon. *Dagstuhl Reports*, 2004.

[95] J.-M. Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*, pages 262–271. Citeseer, 2004.

[96] J.-M. Favre. Languages evolve too! changing the software time scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 33–42. IEEE, 2005.

[97] J.-M. Favre, D. Gasevic, R. Lämmel, and E. Pek. Empirical language analysis in software linguistics. In *International Conference on Software Language Engineering*, pages 316–326. Springer, 2010.

[98] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[99] R. B. Findler, M. Flatt, and M. Felleisen. Semantic Casts: Contracts and Structural Subtyping in a Nominal World. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, pages 365–389. Springer, 2004.

[100] O. Finot, J. Mottu, G. Sunyé, and T. Degueule. Using meta-model coverage to qualify test oracles. In *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, 2013. URL http://ceur-ws.org/Vol-1077/amt13_submission_3.pdf.

[101] M. Flatt. Creating languages in racket. *Communications of the ACM*, 55(1):48–56, 2012.

[102] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In *International Conference on Model Driven Engineering Languages and Systems*, pages 7–15. Springer, 2007.

[103] F. Fleurey, B. Morin, A. Solberg, and O. Barais. Mde to manage communications with and between resource-constrained systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2011.

[104] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[105] M. Fowler. Fluent interface. http://martinfowler.com/bliki/FluentInterface.html, 2005.

[106] M. Fowler. Language workbenches: The killer-app for domain specific languages, 2005. URL http://martinfowler.com/articles/languageWorkbench.html.

[107] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[108] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[109] G. Gabrysiak, H. Giese, A. Lüders, and A. Seibel. How can metamodels be used flexibly. In *Proceedings of ICSE 2011 workshop on flexible modeling tools, Waikiki/Honolulu*, volume 22, 2011.

[110] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[111] K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Model Driven Architecture-Foundations and Applications*, pages 34–49. Springer, 2009.

[112] J. Garcia and O. Díaz. Adaptation of Transformations to Metamodel Changes. In *Desarollo de Software Dirigido por Modelos*, pages 1–9, 2010.

[113] D. Gasević, N. Kaviani, and M. Hatala. On Metamodeling in Megamodels. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, pages 91–105, 2007.

[114] A. Gill. *Introduction to the Theory of Finite-state Machines*, volume 16. McGraw-Hill New York, 1962.

[115] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[116] M. Gouseti, C. Peters, and T. van der Storm. Extensible language implementation with object algebras (short paper). In *Proceedings of the 13th International Conference on Generative Programming: Concepts & Experiences (GPCE'14)*, pages 25–28. ACM, 2014.

[117] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Monticore: a framework for the development of textual domain specific languages. In *Companion of the 30th international conference on Software engineering*, pages 925–926. ACM, 2008.

[118] B. Gruschko, D. Kolovos, and R. Paige. Towards synchronizing models with evolving metamodels. In *Proceedings of the International Workshop on Model-Driven Software Evolution*, 2007.

[119] C. Guy, B. Combemale, S. Derrien, J. R. Steel, and J.-M. Jézéquel. On model subtyping. In *Modelling Foundations and Applications*, pages 400–415. Springer, 2012.

[120] A. Haber, M. Look, A. N. Perez, P. M. S. Nazari, B. Rumpe, S. Volkel, and A. Wortmann. Integration of heterogeneous modeling languages via extensible and composable language components. In *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*, pages 19–31. IEEE, 2015.

[121] M. K. Hamiaz, M. Pantel, X. Thirioux, and B. Combemale. Correct-by-construction model driven engineering composition operators. *Formal Aspects of Computing*, pages 1–32, 2016.

[122] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming (SCP)*, 8(3):231–274, 1987.

[123] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of" semantics"? *Computer*, 37(10):64–72, 2004.

[124] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *ACM Sigplan Notices*, 24 (11):43–75, 1989.

[125] R. Heim, P. M. S. Nazari, B. Rumpe, and A. Wortmann. Compositional language engineering using generated, extensible, static type-safe visitors. In *European Conference on Modelling Foundations and Applications*, pages 67–82. Springer, 2016.

[126] G. T. Heineman and W. T. Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, page 5, 2001.

[127] P. R. Henriques, M. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the lisa system. *IEE Proceedings-Software*, 152(2):54–69, 2005.

[128] M. Herrmannsdoerfer. Cope–a workbench for the coupled evolution of metamodels and models. In *Software Language Engineering*, pages 286–295. Springer, 2010.

[129] M. Herrmannsdoerfer, S. Benz, and E. Juergens. Automatability of coupled evolution of metamodels and models in practice. In *Model Driven Engineering Languages and Systems*, pages 645–659. Springer, 2008.

[130] M. Herrmannsdoerfer, S. Benz, E. Juergens, et al. Cope: A language for the coupled evolution of metamodels and models. In *1st International Workshop on Model Co-Evolution and Consistency Management*, 2008.

[131] M. Herrmannsdoerfer, S. Benz, and E. Juergens. Cope-automating coupled evolution of metamodels and models. In *ECOOP 2009–Object-Oriented Programming*, pages 52–76. Springer, 2009.

[132] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth. *An extensive catalog of operators for the coupled evolution of metamodels and models.* Springer, 2010.

[133] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[134] K. Holldobler, B. Rumpe, and I. Weisemoller. Systematically deriving domain-specific transformation languages. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 136–145. IEEE, 2015.

[135] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[136] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 633–642. IEEE, 2011.

[137] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 471–480, 2011.

[138] J. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[139] J. D. Ichbiah, R. Firth, P. N. Hilfinger, O. Roubine, M. Woodger, J. G. Barnes, J.-R. Abrial, J.-L. Gailly, J.-C. Heliard, H. F. Ledgard, et al. *Reference manual for the Ada programming language*. Castle House Publications Limited, 1983.

[140] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. Lua-an extensible extension language. *Software Practice & Experience*, 26(6): 635–652, 1996.

[141] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[142] P. Inostroza and T. van der Storm. Modular interpreters with implicit context propagation. *Computer Languages, Systems & Structures*, 2016.

[143] C. Jeanneret, M. Glinz, and B. Baudry. Estimating Footprints of Model Operations. In *Proceedings of the 33rd Internal Conference on Software Engineering (ICSE'11)*, 2011.

[144] JetBrains. Meta-Programming System. https://www.jetbrains.com/mps/.

[145] J. Jézéquel, D. Méndez-Acuña, T. Degueule, B. Combemale, and O. Barais. When systems engineering meets software language engineering. In *Proceedings of the Fifth International Conference on Complex Systems Design & Management (CSD&M 2014), Paris, France, November 12-14, 2014*, pages 1–13, 2014. doi: 10.1007/978-3-319-11617-4_1.

[146] J.-M. Jézéquel. Model driven design and aspect weaving. *Software & Systems Modeling*, 7(2):209–218, 2008.

[147] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Software and Systems Modeling (SoSyM)*, 14(2): 905–920, 2015.

[148] S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[149] A. Johnstone, E. Scott, and M. van den Brand. Modular grammar specification. *Science of Computer Programming*, 87:23–43, 2014.

[150] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming (SCP)*, 72(1): 31–39, 2008.

[151] G. Kahn. Natural semantics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer, 1987.

[152] G. Karsai, M. Maroti, Á. Lédeczi, J. Gray, and J. Sztipanovits. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, 2004.

[153] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel. Design guidelines for domain specific languages. In *Proceedings of the 9th Workshop on Domain-Specific Modeling (DSM'09)*, 2009.

[154] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.

[155] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *Proceedings of the 25th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 444–463. ACM, 2010.

[156] S. Kell. In search of types. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 227–241. ACM, 2014.

[157] M. Kerboeuf and J.-P. Babau. A DSML for Reversible Transformations. In *Proceedings of the 11th Workshop on Domain-Specific Modeling (DSM'11)*, pages 1–6, 2011.

[158] A. A. Khwaja and J. E. Urban. Syntax-directed editing environments: Issues and features. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, pages 230–237. ACM, 1993.

[159] A. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[160] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.

[161] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[162] D. S. Kolovos, R. F. Paige, and F. A. Polack. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, volume 20062, page 200, 2006.

[163] D. S. Kolovos, R. F. Paige, and F. A. Polack. Merging models with the epsilon merging language (EML). In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*, pages 215–229. Springer, 2006.

[164] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.

[165] D. S. Kolovos, N. D. Matragkas, H. H. Rodríguez, and R. F. Paige. Programmatic muddle management. In *XM@ MoDELS*, pages 2–10, 2013.

[166] A. Königs. Model transformation with triple graph grammars. In *Model Transformations in Practice Satellite Workshop of MODELS*, page 166, 2005.

[167] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, pages 297–315. Springer, 2008.

[168] H. Krahn, B. Rumpe, and S. Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.

[169] T. Kühn and W. Cazzola. Apples and oranges: comparing top-down and bottom-up language product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 50–59. ACM, 2016.

[170] T. Kühn, W. Cazzola, and D. M. Olivares. Choosy and picky: configuration of language product lines. In *Proceedings of the 19th International Conference on Software Product Line*, pages 71–80. ACM, 2015.

[171] T. Kühne. On Model Compatibility with Referees and Contexts. *Software and Systems Modeling (SoSyM)*, 12(3):475–488, 2013.

[172] A. Kunert. Semi-automatic generation of metamodels and models from grammars and programs. *Electronic Notes in Theoretical Computer Science*, 211:111–119, 2008.

[173] A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Retschitzegger, W. Schwinger, and J. Schonbock. Consistent co-evolution of models and transformations. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 116–125. IEEE, 2015.

[174] F. Latombe, X. Crégut, B. Combemale, J. Deantoni, and M. Pantel. Weaving Concurrency in Executable Domain-specific Modeling Languages. In *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 125–136. ACM, 2015.

[175] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, page 114, 2001.

[176] Á. Lédeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti. On metamodel composition. In *Control Applications, 2001.(CCA'01). Proceedings of the 2001 IEEE International Conference on*, pages 756–760. IEEE, 2001.

[177] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *European Symposium on Programming*, pages 219–234. Springer, 1996.

[178] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[179] D. H. Lorenz and B. Rosenan. Cedalion: a language for language oriented programming. In *ACM SIGPLAN Notices*, volume 46, pages 733–752. ACM, 2011.

[180] T. Mayerhofer and M. Wimmer. The ttc 2015 model execution case. *8th Transformation Tool Contest, CEUR*, 2015.

[181] J. McCarthy. Towards a mathematical science of computation. In *Program Verification*, pages 35–56. Springer, 1993.

[182] S. J. Mellor, M. Balcer, and I. Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[183] D. Mendez, A. Etien, A. Muller, and R. Casallas. Towards Transformation Migration After Metamodel Evolution. In *Model and Evolution Workshop*, Model and Evolution Workshop, 2010.

[184] D. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin, B. Baudry, and G. Le Guernic. Reverse-engineering reusable language modules from legacy domain-specific languages. In *International Conference on Software Reuse*, pages 368–383. Springer, 2016.

[185] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry. Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures*, 2016. ISSN 1477-8424. doi: http://dx.doi.org/10.1016/j.cl.2016.09.004.

[186] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[187] M. Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9): 2451–2464, 2013.

[188] M. Mernik and V. Zumer. Reusability of formal specifications in programming language description, 1997.

[189] M. Mernik, M. Lenic, E. Avdicauševic, and V. Zumer. Multiple attribute grammar inheritance. *Informatica*, 24(3):319–328, 2000.

[190] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Lisa: An interactive environment for programming language development. In *International Conference on Compiler Construction*, pages 1–4. Springer, 2002.

[191] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37(4): 316–344, 2005.

[192] B. Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[193] B. Meyers and H. Vangheluwe. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):1223–1246, 2011.

[194] P. D. Mosses. The varieties of programming language semantics and their uses. In *Perspectives of System Informatics*, pages 165–190. Springer, 2001.

[195] P. D. Mosses. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:195–228, 2004.

[196] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229 (4):49–66, 2009.

[197] S. Mustafiz, B. Barroca, C. Gomes, and H. Vangheluwe. Towards modular language design using language fragments: The hybrid systems case study. In *Information Technology: New Generations*, pages 785–797. Springer International Publishing, 2016.

[198] A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai. Automatic domain model migration to manage metamodel evolution. In *MoDELS*, volume 9, pages 706–711. Springer, 2009.

[199] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[200] B. C. d. S. Oliveira. Modular visitor components. In *European Conference on Object-Oriented Programming*, pages 269–293. Springer, 2009.

[201] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses – Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*, pages 2–27. Springer, 2012.

[202] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP 2014–Object-Oriented Programming*, pages 105–130. Springer, 2014.

[203] OMG. Interface Definition Language. http://www.omg.org/spec/IDL/.

[204] OMG. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/spec/MOF/2.0/, 2006.

[205] OMG. Business Process Model and Notation (BPMN) 2.0. http://www.omg.org/spec/BPMN/2.0/, 2011.

[206] OMG. UML Profile For MARTE: Modeling and Analysis of Real-Time Embedded Systems. http://www.omg.org/spec/MARTE/, 2011.

[207] OMG. Unified Modeling Language (UML), Infrastructure Specification. http://www.omg.org/spec/UML/, 2011.

[208] OMG. Common Object Request Broker Architecture (CORBA) Specification 3.3. http://www.omg.org/spec/CORBA/3.3/, 2012.

[209] OMG. Object Constraint Language (OCL). http://www.omg.org/spec/OCL/, 2012.

[210] OMG. Systems Modeling Language (SysML). http://www.omg.org/spec/SysML/, 2015.

[211] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) 1.3. http://www.omg.org/spec/QVT/1.3, 2016.

[212] J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27(2): 196–255, 1995.

[213] R. F. Paige, D. S. Kolovos, and F. A. Polack. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, 96:396–416, 2014.

[214] R. F. Paige, N. Matragkas, and L. M. Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, 2016.

[215] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[216] D. L. Parnas, J. E. Shore, and D. Weiss. Abstract types defined as classes of variables. In *Proceedings of the 1976 Conference on Data : Abstraction, definition and structure*, pages 149–154. ACM, 1976.

[217] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[218] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara. Pattern-based development of domain-specific modelling languages. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 166–175. IEEE, 2015.

[219] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, 1981.

[220] C. B. Poulsen and P. D. Mosses. Generating specialized interpreters for modular structural operational semantics. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 220–236. Springer, 2013.

[221] L. Renggli and T. Gîrba. Why smalltalk wins the host languages shootout. In *Proceedings of the International Workshop on Smalltalk Technologies*, pages 107–113. ACM, 2009.

[222] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11): 60–67, 2009.

[223] E. Roche and Y. Schabes. *Finite-state Language Processing*. MIT press, 1997.

[224] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*, pages 127–136. ACM, 2010.

[225] L. Rose, E. Guerra, J. De Lara, A. Etien, D. Kolovos, and R. Paige. Genericity for model management operations. *Software & Systems Modeling*, 12(1):201–219, 2013.

[226] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model Migration with Epsilon Flock. In *Proceedings of the 4th International Conference on Model Transformation (ICMT'10)*, pages 184–198, 2010.

[227] G. Roşu and T. F. Şerbănuţă. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[228] J. Sánchez Cuadrado and J. García Molina. Approaches for Model Transformation Reuse: Factorization and Composition. In *Proceedings of the 1st International Conference on Model Transformations (ICMT'08)*, pages 168–182, 2008.

[229] J. Saraiva. Component-based programming for higher-order attribute grammars. In *International Conference on Generative Programming and Component Engineering*, pages 268–282. Springer, 2002.

[230] D. Schmidt. Denotational semantics: A methodology for language development. 1997.

[231] D. C. Schmidt. Model-driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[232] A. Schürr. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[233] E. Seidewitz. UML with meaning: executable modeling in foundational UML and the alf action language. In *Proceedings of the 2014 Annual Conference on High Integrity Language Technology (HILT'14)*, pages 61–68. ACM, 2014.

[234] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model Pruning. In *Proceedings of the 12th Internal Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, 2009.

[235] S. Sendall and W. Kozaczynski. Model transformation the heart and soul of model-driven software development. *IEEE Software*, 2003.

[236] J. Siegel. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons New York, NY, USA:, 2000.

[237] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *Proceedings of the 21th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'06)*, pages 451–464. ACM, 2006.

[238] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of systems and software*, 56(1):91–99, 2001.

[239] J. Steel and J.-M. Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.

[240] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[241] W. Sun, B. Combemale, S. Derrien, and R. B. France. Using Model Types to Support Contract-aware Model Substitutability. In *Proceedings of the 9th European Conference on Modelling Fundations and Applications (ECMFA'13)*, pages 118–133. Springer, 2013.

[242] A. M. Şutîi, T. Verhoeff, and M. Van den Brand. Modular multilevel metamodeling with metamod. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 212–217. ACM, 2016.

[243] E. Syriani and H. Ergin. Operational semantics of uml activity diagram: An application in project management. In *Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE*, pages 1–8. IEEE, 2012.

[244] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. Openjava: A class-based macro system for java. In *Workshop on Reflection and Software Engineering*, pages 117–133. Springer, 1999.

[245] T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24 (9):563–573, 1981.

[246] The Metamodel Zoos. The Metamodel Zoos, last access: June 2016. URL http://web.emn.fr/x-info/atlanmod/index.php?title=Zoos.

[247] J.-P. Tolvanen and S. Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[248] J.-P. Tolvanen and M. Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93. ACM, 2003.

[249] E. Vacchi and W. Cazzola. Neverlang: a framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43: 1–40, 2015.

[250] E. Vacchi, W. Cazzola, S. Pillay, and B. Combemale. Variability support in domain-specific language development. In *Software Language Engineering*, pages 76–95. Springer, 2013.

[251] M. Van den Brand, A. van Deursen, J. Heering, H. De Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, et al. The asf+sdf meta-environment: A component-based language development environment. In *International Conference on Compiler Construction*, pages 365–370. Springer, 2001.

[252] M. Van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework—tool demonstration—. *Electronic Notes in Theoretical Computer Science*, 141(4):161–165, 2005.

[253] T. van der Storm. *The Rascal language workbench*. CWI. Software Engineering [SEN], 2011.

[254] A. Van Deursen and P. Klint. Little languages: little maintenance? *Journal of software maintenance*, 10(2):75–92, 1998.

[255] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[256] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1): 39–54, 2010.

[257] D. Varró and A. Pataricza. Generic and Meta-transformations for Model Transformation Engineering. In *Proceedings of the 7th International Conference on UML Modelling Languages and Applications (UML'04)*, pages 290–304, 2004.

[258] V. Vergu, P. Neron, and E. Visser. Dynsem: A dsl for dynamic semantics specification. Technical report, Delft University of Technology, Software Engineering Research Group, 2015.

[259] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, and G. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 95–111. ACM, 2014.

[260] M. Voelter. Language and ide modularization and composition with mps. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 383–430. Springer, 2011.

[261] M. Voelter. *Generic tools, specific languages*. TU Delft, Delft University of Technology, 2014.

[262] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, 16, 2010.

[263] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[264] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace, 2013.

[265] M. Voelter, B. Kolb, and J. Warmer. Projecting a modular future. *IEEE Software*, 32(5), 2014.

[266] M. Völter and E. Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–304. ACM, 2010.

[267] W3C. XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, May 2001.

[268] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, pages 600–624. Springer, 2007.

[269] G. H. Wachsmuth, G. D. Konat, and E. Visser. Language design with the spoofax language workbench. *Software, IEEE*, 31(5):35–43, 2014.

[270] P. Wadler. The expression problem. *Java-genericity mailing list*, 1998.

[271] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[272] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, pages 439–449. IEEE Press, 1981.

[273] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[274] M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *International Conference on Model Driven Engineering Languages and Systems*, pages 159–168. Springer, 2005.

[275] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[276] N. Wirth. Modula: A language for modular multiprogramming. *Software: Practice and Experience*, 7(1):1–35, 1977.

[277] J. Zhang, Y. Lin, and J. Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development*, pages 199–217. Springer, 2005.

[278] A. Zolotas, N. Matragkas, S. Devlin, D. S. Kolovos, and R. F. Paige. Type inference in flexible model-driven engineering. In *European Conference on Modelling Foundations and Applications*, pages 75–91. Springer International Publishing, 2015.

Appendix A

Listings

```
1  package fsmfamily
2
3  language FlatFsmRtc {
4    syntax "FlatFsm.ecore"
5    with semantics.flat.StateMachineAspect
6    with semantics.flat.StateAspect
7    exactType FlatFsmRtcMT
8  }
9
10 language FlatFsmSimultaneous {
11   syntax "FlatFsm.ecore"
12   with semantics.flat.simultaneous.StateMachineAspect
13   with semantics.flat.simultaneous.StateAspect
14   exactType FlatFsmSimultaneousMT
15 }
16
17 language TimedFsmRtc {
18   syntax "TimedFsm.ecore"
19   with semantics.timed.StateMachineAspect
20   with semantics.timed.StateAspect
21   with semantics.timed.TransitionAspect
22   exactType TimedFsmRtcMT
23 }
24
25 language TimedFsmSimultaneous {
26   syntax "TimedFsm.ecore"
27   with semantics.timed.simultaneous.StateMachineAspect
28   with semantics.timed.simultaneous.StateAspect
29   with semantics.timed.simultaneous.TransitionAspect
30   exactType TimedFsmSimultaneousMT
31 }
32
33 language CompositeFsmRtc {
34   syntax "CompositeFsm.ecore"
35   with semantics.composite.StateMachineAspect
36   with semantics.composite.StateAspect
37   exactType CompositeFsmRtcMT
38 }
39
40 language CompositeFsmSimultaneous {
41   syntax "CompositeFsm.ecore"
42   with semantics.composite.simultaneous.StateMachineAspect
```

```
43    with semantics.composite.simultaneous.StateAspect
44    exactType CompositeFsmSimultaneousMT
45  }
46
47  language TimedCompositeFsmRtc {
48    syntax "TimedCompositeFsm.ecore"
49    with semantics.timedcomposite.StateMachineAspect
50    with semantics.timedcomposite.StateAspect
51    with semantics.timedcomposite.TransitionAspect
52    exactType TimedCompositeFsmRtcMT
53  }
54
55  language TimedCompositeFsmSimultaneous {
56    syntax "TimedCompositeFsm.ecore"
57    with semantics.timedcomposite.simultaneous.StateMachineAspect
58    with semantics.timedcomposite.simultaneous.StateAspect
59    with semantics.timedcomposite.simultaneous.TransitionAspect
60    exactType TimedCompositeFsmSimultaneousMT
61  }
```

Listing A.1: The complete Melange specification describing the eight variants of finite-state machine languages presented in Chapter 5

```
1   @Aspect(className = OpaqueAction)
2   class OpaqueActionAspect extends ActivityNodeAspect {
3     public OperationDef service
4
5     def void execute(Context c) {
6       c.output.executedNodes.add(_self)
7       val fact = ActivitydiagramFactory::eINSTANCE
8       if (_self.service !== null) {
9         // 1- Translate the environment from ActivityDiagram to IDL
10        val wrappedEnv = new Environment => [
11          _self.service.parameters
12          .filter[
13            #[ParameterMode::PARAM_IN, ParameterMode::PARAM_INOUT]
14            .contains(direction)
15          ]
16          .forEach[p |
17            val find = _self.activity.locals.findFirst[name == p.identifier]
18            putVariable(p.identifier, _self.getValue(find?.currentValue) ?: null)
19          ]
20        ]
21
22        // 2- Execute the service
23        _self.service.execute(wrappedEnv)
24
25        // 3- Translate back the environment from IDL to the ActivityDiagram
26        _self.service.parameters
27        .filter[
28          #[ParameterMode::PARAM_OUT, ParameterMode::PARAM_INOUT]
29          .contains(direction)
30        ]
31        .forEach[p |
32          val updated = _self.activity.locals.findFirst[name == p.identifier]
33          val retInteger = new Integer(Double::parseDouble(
34            wrappedEnv.getVariable(p.identifier).toString
35          ) as int)
36
37          if (updated !== null)
```

```
38            updated.currentValue = fact.createIntegerValue => [
39              value = retInteger
40            ]
41          else
42            _self.activity.locals += fact.createIntegerVariable => [
43              name = p.identifier
44              currentValue = fact.createIntegerValue => [
45                value = retInteger
46              ]
47            ]
48        ]
49      }
50      _self.expressions.forEach[e|e.execute(c)]
51      _self.sendOffers(_self.takeOfferdTokens)
52    }
53
54    def Object getValue(Value v) {
55      return
56        switch (v) {
57          IntegerValue: v.value as double
58          BooleanValue: v.value
59          default: null
60        }
61    }
62  }
```

Listing A.2: The `OpaqueActionAspect` linking ActivityDiagram's `OpaqueAction` to IDL's `OperationDef`

```
1  package fr.inria.diverse.minifsm
2
3  language MiniFsm {
4    syntax "MiniFsm.ecore"
5    with minifsm.aspects.FinalStateAspect
6    with minifsm.aspects.TransitionAspect
7    with minifsm.aspects.StateAspect
8    with minifsm.aspects.FSMAspect
9  }
10
11 language MiniActionLang {
12   syntax "MiniActionLang.ecore"
13   syntax "MiniActionLangRuntime.ecore"
14   with minilang.aspects.*
15 }
16
17 language MelangedLang inherits MiniFsm {
18   // Slice away the If and While constructs of MiniActionLang
19   // and use MiniFsm instead to express the control flow
20   slice- MiniActionLang on ["If", "While"]
21     renaming { "minilang" to "minifsm" }
22
23   // Glue together MiniFsm and MiniActionLang
24   with fr.inria.diverse.melanger.FSMGlue
25   with fr.inria.diverse.melanger.StateGlue
26   with fr.inria.diverse.melanger.TransitionGlue
27 }
```

Listing A.3: The complete Melange specification of the `MiniFsm` language presented in Chapter 7

```
@Aspect(className = FSM)
class FSMGlue extends FSMAspect {
  @Containment public Context context

  override void execute(EList<String> events) {
    val eventIt = events.iterator

    if(eventIt.hasNext)
      _self.currentEvent = eventIt.next

    _self.currentState = _self.initialState

    while (_self.currentState !== null) {
      _self.currentState.execute()
      if (_self.currentState instanceof FinalState)
        _self.currentState = null
      else {
        val candidate = _self.transitions.findFirst[
          input === _self.currentState && isActivated]
        _self.currentState = candidate?.output
      }
    }
  }
}

@Aspect(className = State)
class StateGlue extends StateAspect {
  // Associate each State to a Block of code of MiniActionLang
  @Containment public Block block

  // Override MiniFsm's implementation of State::execute()
  // to delegate to the interpreter of MiniActionLang
  @OverrideAspectMethod
  override void execute() {
    _self.block?.statement?.forEach[execute(_self.fsm.context)]
  }
}

@Aspect(className = Transition)
class TransitionGlue extends TransitionAspect {
  // Use MiniActionLang's BooleanExpression as a guard
  // for each transition of MiniFsm
  @Containment public BooleanExpression expression

  @OverrideAspectMethod
  override boolean isFireable() {
    return _self.expression == null ||
           _self.expression.eval(_self.fsm.context)
  }
}
```

Listing A.4: The aspects defining the glue between the languages of Listing A.3

```
package simpleaspects.fsm.adapters.fsmmt.fsm;

import fr.inria.diverse.melange.adapters.EObjectAdapter;
import org.eclipse.emf.ecore.EClass;
import simpleaspects.fsm.fsm.Transition;
```

```
 6  import simpleaspects.fsmmt.fsm.State;
 7  import simpleaspects.fsm.adapters.fsmmt.FsmMTAdaptersFactory;
 8
 9  public class TransitionAdapter extends EObjectAdapter<Transition>
10    implements simpleaspects.fsmmt.fsm.Transition {
11    private FsmMTAdaptersFactory adaptersFactory;
12
13    public TransitionAdapter() {
14      super(FsmMTAdaptersFactory.getInstance());
15      adaptersFactory = FsmMTAdaptersFactory.getInstance();
16    }
17    @Override
18    public String getInput() {
19      return adaptee.getInput();
20    }
21    @Override
22    public void setInput(final String o) {
23      adaptee.setInput(o);
24    }
25    @Override
26    public String getOutput() {
27      return adaptee.getOutput();
28    }
29    @Override
30    public void setOutput(final String o) {
31      adaptee.setOutput(o);
32    }
33    @Override
34    public State getSource() {
35      return (State) adaptersFactory.createAdapter(adaptee.getSource(), eResource);
36    }
37    @Override
38    public void setSource(final State o) {
39      if (o != null)
40        adaptee.setSource(
41      ((simpleaspects.fsm.adapters.fsmmt.fsm.StateAdapter) o).getAdaptee());
42      else adaptee.setSource(null);
43    }
44    @Override
45    public State getTarget() {
46      return (State) adaptersFactory.createAdapter(adaptee.getTarget(), eResource);
47    }
48    @Override
49    public void setTarget(final State o) {
50      if (o != null)
51        adaptee.setTarget(
52      ((simpleaspects.fsm.adapters.fsmmt.fsm.StateAdapter) o).getAdaptee());
53      else adaptee.setTarget(null);
54    }
55    @Override
56    public void fire() {
57      simpleaspects.fsm.aspects.ExecutableTransitionAspect.fire(adaptee);
58    }
59    protected final static String INPUT_EDEFAULT = null;
60    protected final static String OUTPUT_EDEFAULT = null;
61    @Override
62    public EClass eClass() {
63      return simpleaspects.fsmmt.fsm.FsmPackage.eINSTANCE.getTransition();
64    }
65    @Override
66    public Object eGet(final int featureID, final boolean resolve,
67      final boolean coreType) {
```

```
68      switch (featureID) {
69        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__SOURCE:
70          return getSource();
71        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__TARGET:
72          return getTarget();
73        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__INPUT:
74          return getInput();
75        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__OUTPUT:
76          return getOutput();
77      }
78      return super.eGet(featureID, resolve, coreType);
79    }
80    @Override
81    public boolean eIsSet(final int featureID) {
82      switch (featureID) {
83        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__SOURCE:
84          return getSource() != null;
85        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__TARGET:
86          return getTarget() != null;
87        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__INPUT:
88          return getInput() != INPUT_EDEFAULT;
89        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__OUTPUT:
90          return getOutput() != OUTPUT_EDEFAULT;
91      }
92      return super.eIsSet(featureID);
93    }
94    @Override
95    public void eSet(final int featureID, final Object newValue) {
96      switch (featureID) {
97        case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__SOURCE:
98          setSource(
99          (simpleaspects.fsmmt.fsm.State)
100          newValue);
101         return;
102       case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__TARGET:
103         setTarget(
104         (simpleaspects.fsmmt.fsm.State)
105          newValue);
106         return;
107       case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__INPUT:
108         setInput(
109         (java.lang.String)
110          newValue);
111         return;
112       case simpleaspects.fsmmt.fsm.FsmPackage.TRANSITION__OUTPUT:
113         setOutput(
114         (java.lang.String)
115          newValue);
116         return;
117     }
118     super.eSet(featureID, newValue);
119   }
120 }
```

Listing A.5: Adapter generated between the `Transition` meta-class of the `Fsm` language and the `Transition` object type of the `FsmMT` model type

Abstract

Following the principles of Model-Driven Engineering and Language-Oriented Programming, Domain-Specific Languages (DSLs) are now developed in numerous domains to address specific concerns in the development of complex systems. However, despite many advances in Software Language Engineering, DSLs and their tooling still suffer from substantial development costs which hamper their successful adoption in the industry.

We identify two main challenges to be addressed. First, the proliferation of independently developed and constantly evolving DSLs raises the problem of interoperability between similar languages and environments. Language users must be given the flexibility to open and manipulate their models using different variants and versions of various environments and services to foster collaboration in the development of complex systems. Second, since DSLs and their environments suffer from high development costs, tools and methods must be provided to assist language designers and mitigate development costs.

In this thesis, we address these challenges through three interconnected contributions. First, we propose the notion of *language interface*. Using language interfaces, one can vary or evolve the implementation of a DSL while retaining the compatibility with the services and environments defined on its interface. Then, we present a mechanism, named *model polymorphism*, for manipulating models through different language interfaces. Model polymorphism opens up the possibility to safely manipulate models using different modeling environments and services. Finally, we propose a meta-language that enables language designers to reuse legacy DSLs, compose them, extend them, and customize them to meet new requirements. This approach relies on language interfaces to provide a reasoning layer for ensuring the structural correctness of composed DSLs and their safe manipulation.

We implement all our contributions in a new language workbench named Melange. Melange supports the modular definition of DSLs, and the interoperability of their environments. Melange is seamlessly integrated with the *de facto* standard Eclipse Modeling Framework (EMF) and provides model polymorphism to any EMF-based tool of the Eclipse modeling ecosystem. Using Melange, we show how to reuse tools and services over various language families (four versions of the Unified Modeling Language, and a family of statechart languages), and how to flexibly manipulate their models. We also show how Melange eases the development of new DSLs by designing a new modeling language for Internet of Things systems as an assembly of various independently developed languages.

Different perspectives directly stem from the contributions presented in this thesis. In particular, we discuss how our contributions constitutes a first step towards component-based language engineering and viewpoints engineering.